Understanding CGP-IP

Benjamin Tregoning
n8589283

QUT Final Year Thesis

2015

The CGP-IP code was altered to work in Linux. Some functionality had to be removed but it had no effect on its core. A filter maker was also created and successfully used in the analysis process. Analysis was also done on two main points. The similarities between filters and the speed of certain functions
**Contents**

1. List of Figures ............................................................................................................... 3
2. List of Tables .................................................................................................................. 4
3. Statement of Original Authorship ................................................................................. 5
4. Introduction .................................................................................................................... 6
5. Related Work .................................................................................................................. 10
6. Methodology ................................................................................................................ 13

4.1 Setting Up CGP-IP ..................................................................................................... 13

4.1.1 Datasets .................................................................................................................. 16

4.1.1.1 Cow Cup .............................................................................................................. 16
4.1.1.2 Container .......................................................................................................... 17
4.1.1.3 Tea Box .............................................................................................................. 17
4.1.1.4 Rep/Rot .............................................................................................................. 17
4.1.1.5 Weizmann Segmentation Database ................................................................... 18

4.1.2 Main Class, Program.cs ....................................................................................... 18

4.1.2.1 Program.cs function Go(string[] args) ................................................................. 18
4.1.2.2 Program.cs function StartServer(bool Spawn) ..................................................... 19
4.1.2.3 Program.cs function StartClient() ...................................................................... 20
4.1.2.4 Fitness Function Class FitnessFunction.cs ......................................................... 20
4.1.2.5 Program.cs function TestPopulation(FitnessFunction FitFunc, CGPPopulation Pop) .................................................................................................................. 20

4.1.3 Matthews Correlation Coefficient (MCC) and Fitness Function ......................... 21

4.1.4 Program Flow ......................................................................................................... 22

4.1.4.1 CGP-IP functionality .......................................................................................... 24
4.1.4.2 FilterMaker Functionality ................................................................................ 27

4.1.5 Local Minimum ..................................................................................................... 28
7. Progress .......................................................................................................................... 29
   5.1 Semester One ................................................................................................................. 29
   5.2 Semester Two ............................................................................................................... 31
      5.2.1 Translating CGP-IP into the Linux Platform ......................................................... 31
      5.2.2 Re-Purposing the Graph Runner to Create a Filter ........................................... 34
8. Experiments .................................................................................................................. 36
   6.1 Experiment One .......................................................................................................... 36
   6.2 Experiment Two ......................................................................................................... 36
   6.3 Experiment Three ..................................................................................................... 36
9. Experiment Results ...................................................................................................... 38
   7.1 Experiment One .......................................................................................................... 38
      7.1.1 Container Dataset ............................................................................................... 38
      7.1.2 Cow Cup Dataset ............................................................................................... 44
      7.1.3 Rep/Rot Dataset – Calculator (RepCalc) ............................................................ 50
   7.2 Experiment Two ........................................................................................................ 56
      7.2.1 RepPaper ............................................................................................................ 56
      7.2.2 TeaBox ................................................................................................................ 58
   7.3 Experiment Three ..................................................................................................... 59
      7.3.1 Filter One ............................................................................................................. 59
      7.3.2 Filter Two ........................................................................................................... 63
      7.3.3 Filter Three ........................................................................................................ 66
10. Discussion ................................................................................................................... 69
11. Conclusion .................................................................................................................... 71
12. Bibliography ................................................................................................................ 72
1. List of Figures

Figure 1: Example CGP output ................................................................. 8
Figure 2: An example parameters file .................................................... 15
Figure 3: Flow chart of CGP-IP ............................................................... 22
Figure 4: Picture of IDE showing the correct and incorrect MCC values .... 33
Figure 5: Filter One best individual ....................................................... 38
Figure 6: Filter Two best individual ........................................................ 39
Figure 7: Filter Three best individual ..................................................... 40
Figure 8: Filter Four best individual ...................................................... 41
Figure 9: Filter Five best individual ...................................................... 42
Figure 10: Filter One best individual .................................................... 44
Figure 11: Filter Two best individual .................................................... 45
Figure 12: Filter Three best individual .................................................. 46
Figure 13: Filter Four best individual .................................................... 47
Figure 14: Filter One best individual .................................................... 50
Figure 15: Filter Two best individual .................................................... 51
Figure 16: Filter Three best individual .................................................. 52
Figure 17: Filter Four best individual .................................................... 53
Figure 18: Filter Five best individual .................................................... 54
Figure 19: Left, filter output. Right, evolution output ................................ 57
Figure 20: Left, filter output. Right, evolution output ................................ 58
Figure 21: TeaBox output ..................................................................... 58
Figure 22: Difference between Gaussian and bilateral smoothing. Source:  http://ftparmy.com/64316-bilateral-filter.html ............................................. 60
Figure 23: Left, Slow filter output. Right, TeaBox input ............................ 60
Figure 24: Left, Fast filter output. Right, TeaBox input ............................. 62
Figure 25: Left, Slow filter output. Right, RepMetal input ......................... 64
Figure 26: Left, Fast filter output. Right, RepMetal input .......................... 65
Figure 27: Left, Slow filter output. Right, RepCalc input ........................... 66
Figure 28: Top left: Original output. Top Right: Dilated output. Bottom Left: Eroded output. Bottom Right: Gradient output ......................................................... 67
2. List of Tables

Table 1: Container Function Count ................................................................................. 43
Table 2: Cow Cup function count .................................................................................... 48
Table 3: RepCalc function count ..................................................................................... 55
Table 4: RepPaper Output ................................................................................................. 56
3. Statement of Original Authorship

The work contained in this report has not been previously submitted for a degree or diploma at any other tertiary educational institution. To the best of my knowledge and belief, the project report contains no material previously published or written by another person except where due reference is made.

Signed:_________________________

Date: 23/10/2015
4. Introduction

Image Processing is a very important part of robotics. In order to increase the viability of robotic systems, they need to be able to see and interact with their surroundings. A lot of high end robots (such as the ones in a car factory) do not use any sort of visual aid and instead rely on precision movement to understand their positions and the positions of the parts they are specifically designed for. The future of robotics lies in robots having the ability to be fully aware of their surroundings and to dynamically interact with it. It is for this reason that there is a lot of research in image processing, especially for robots and other automated systems.

One such approach to image processing is Cartesian Genetic Programming for Image Processing (CGP-IP). Genetic Programming (GP) is a systematic method for getting computers to automatically solve a problem starting from a high-level statement of what needs to be done. Specifically, Genetic Programming iteratively transforms a problem by applying analogues of naturally occurring genetic operations.\(^1\) It works using three types of inputs; a given set of possible inputs, a given set of operations and a method (called fitness function) to find its similarity with the expected output (called fitness).

When started, GP randomly selects a certain number of inputs and operations. Each operation and input is called a node and the nodes are connected together to form a genotype. This is done multiple times to create a population of individuals and these populations are called a generation. There can be any number of populations and each population executes the same steps. Each Individual’s genotype is represented by a tree style graph with each node consisting of either an input or an operation. Each individual is then compared to a fitness function, returning a fitness value. The fitness function is a function that determines how well in individual performs and is usually a comparison between the expected output and the output of the individual. The individual with the best fitness is chosen to be the basis of the next generation. The next generation is made by altering (or not altering) the previous best individual.

These alterations are of four different types; reproduction, crossover and mutation. Reproduction is simply copying the individual into the next generation. Crossover is making a new individual by taking one part of an individual and swapping it with another similar
part of a different individual, creating two new individuals. Crossover is more likely to be done when two or more individuals are chosen for the new generation. Mutation is randomly selecting a part of the individual and altering it. For example, a new node may be added or a node gets changed. This procedure is repeated until an individual with a fitness, determined beforehand, that is satisfactory is achieved.

The level of fitness depends on the difficulty of the task needed to be solved. A simple task, such as a program that will add integers together to equal 100 can have a fitness based off the output number. The fitness can be based off whether the number equals 100 and possibly how many additions it needed to achieve it. However, a more difficult task such as developing a process for a robot to navigate a room will have a much more complex fitness calculation as there are many more factors than simple integers.

Along with the possible inputs and operations, the probability of each of the changes happening are also given so the type of change can be determined. The amount of individuals in each generation (called a population) can also be changed.

Cartesian Genetic Programming (CGP) is very similar to Genetic Programming with a few key differences. CGP represents phenotypes of individuals as a group of nodes addressable in a Cartesian coordinate system, as opposed to the traditional tree based method. It is characterised by its encoding of the genotype as a string of integers which represent the functions and connections between nodes, program inputs and program outputs. This approach gives CGP very high generality, allowing it to be used to solve many different problems.
Figure 1 shows what an output for CGP may look like. The genotype, which is represented by the string of integers, shows how each output is calculated. Each number in the rectangles is an input for that node. Underlined numbers represent ignored inputs. Nodes with a dashed outline do not contribute towards the output. In this example, Node 4 is using an AND operation on Node 0 (Input A) and Node 2 (Input C).

Cartesian Genetic Programing for Image Processing (CGP-IP) aims to generate programs that can be used to solve specific problems that arise in image processing. These solutions can make use of colour images, utilising both RGB and HSV channels to use as inputs. GCP-IP returns human readable C# or C++ code based in OpenCV.
With all that is known about CGP-IP, there is a lot that is unknown. This project aims to understand some of the intricacies of the ‘choices’ that the CGP-IP process makes. In regards to Image Processing, an analysis of how often each OpenCV function appears in a phenotype/genotype and how each one affects the given outcomes. It will also aim to refine the processes that are used in CGP-IP. This will be done by analysing the output of several datasets. Each dataset will be run several times and some will be run on different operating systems as well. Comparisons will be made between filters evolved on the same datasets.
5. Related Work

H. Kim et al. developed a system that could be taught to recognise objects, possibly blocked or rotated, in an environment that is cluttered. The main focus of this work was to make the system learn with as little human help as possible. The robot used to train with the 100 household objects that the work attempted to recognise was shaped like a human head, having a similar level of freedom of movement as a human (Moving eyes up and down and moving head left and right). When an object was held a certain distance away from the robot in its field of view, it would focus on that object and save video frames as the object moved and rotated. These images would be the basis of the recognition of the object. The training stage extracts the Gabor-Jet features of the various images retrieved and clusters them to create a vocabulary with 64,000 features. During the recognition of the items, features taken from the testing video are used to search each vocabulary using a kd-tree algorithm. The system could successfully detect the objects with an average accuracy of 78.53%. They believed that these limitations were due to having a lower resolution camera, using several similar objects, specular reflection and motion blur. The results demonstrated the feasibility of using this approach and improving.

Image segmentation is the act of grouping pixels in an image based on the objects captured in that image. Shirakawa and Nagao developed an approach for evolutionary image segmentation based on multiobjective clustering. The approach did not use CGP as its basis, but another evolutionary approach called Multiobjective Clustering with automatic K-Determination (MOCK), an altered form of Multiobjective Evolutionary Algorithm (MOEA) to evolve an algorithm that would successfully segment images. This process is quite similar to CGP as it is also an evolutionary method. It outputs a genotype that can be decoded to form a phenotype that shows every region found in the image. It is however specifically designed to create a segmentation algorithm and so it uses techniques specific to that strand.

J. Liu and Y. Yang developed a segmentation algorithm for colour images by drastically lowering the resolution of an image and doing a coarse segmentation and then refining the segmentation by more closely inspecting the segmentation at a certain point. The refining of the segmentation is done by changing the configuration of a quadtree that is first developed using the low resolution. A quadtree is a data structure that represents a two dimensional
image of size $2^N \times 2^N$ in a hierarchy. The tree starts off with the whole image as the root. If the region the node represents is not homogenous, it is divided into four parts. This is repeated until each region is homogenous. The tree is then refined using a relaxation process. The relaxation process uses energy, calculated by the similarities between a node and its parents and its neighbours, to determine whether a node should be split or merged with adjacent nodes or its children. This quadtree is developed for many the many colour spaces of the given image to give a more accurate segmentation of a colour image. This approach managed to deliver satisfactory segmentation of both real and synthetic images; synthetic images being manmade and easily segmented and real images being photos or scanned images.

There have been many works that successfully attempt to apply CGP to a problem. Harding et al.[7] adapted CGP to work with Image Processing. Because CGP is very easily adaptable to a problem due to using an integer to represent an operation, functions from OpenCV were able to be used in nodes. The input nodes were not just the images to be processes, but the various components that make up an image. The image values of Hue, Saturation, Value, Red, Green, and Blue were all possible input nodes. This approach was tested on several domains. The first was the evolution of some basic filters; one for reducing Gaussian noise and one for reducing salt and pepper noise. The output functions compared well to previously published techniques.

CGP-IP was also tested on detecting cancer cells that were undergoing mitosis. This detection is a very important part of combating cancer but the cancer cells are very small and have a wide variety of shapes and so even doctors who specialise in that area may not due to time constraints, CGP-IP was only run 6 times but the evolved detector was able to correctly detect the cells 86% of the time, with 12 false positives and 6 false negatives.

Lastly, CGP-IP was used to develop a filter to detect a certain object among other objects. This was done using images taken from the iCub Humanoid Robot. In the nine training images, the object in question was randomly placed amongst other randomly placed objects. It was also rotated and even positioned behind other objects. The evolved filter was able to detect the object in real time, even when viewed in different light settings and when the object was rotated.
Hardig, Simon and Miller\cite{3} set about to make a robot that would autonomously travel through a room with random layout. The robot was similar to a Kephera Robot, having two wheels for movement and two cameras separated at 20 degrees for vision. The functions used in the learning process were Add, Subtract, Multiply, Divide, Compare, Min, Max, Fixed integer and Input node. Fixed node being a node with no inputs that stores a value between -100 and 100 and input node being the first nodes in the graph – the nodes used for the input. Using CGP to evolve a control sequence for the robot gave a successful run through a maze 70 out of 140 runs, of these 70 successful evolutions, 91\% could run through a different maze with no extra evolution. This showed that the evolved programs had achieved a high degree of generalisation.

Leitner et al.\cite{8} worked on developing a program that lets a humanoid robot learn the representatives of objects in its vision. Its aim was to create a system that would explore an area and using a basic segmentation program, provide inputs for a CGP learner to create efficient object identifiers for the various objects in the room. The basis of this work was to let the robot learn with as little human intervention as possible.
6. Methodology

This section will describe the layout of the code that is GCP-IP. It will describe classes and relevant functions inside those classes as it runs through the main processes. It will also describe the various steps in setting up an image set to be tested. It will also outline changes made during the life of the project.

4.1 Setting Up CGP-IP

When using CGP-IP, it may take a very long time to evolve a program that will work to a satisfactory level. This can be due to the complexity of the problem (finding an object, even if it is partially covered or surrounded by similar objects) or that the solution to the problem is not properly defined (detecting cancer cell mitosis). To help speed up the process, CGP-IP employs a client/server architecture. A central server is set up at the beginning of the CGP-IP program which clients can connect to. The server’s job is to store the best individual made by any of the clients connected to it. It also prints a notification when a client sends a better individual to the server. The client’s job is to carry out the main processing of CGP-IP, taking the individual stored on the server and using it as the basis of the next generation to be evaluated and then sending any better individuals back to the server.

The best part about this kind of architecture is that it can be used to implement distributed computing, making use of not only the machine on which the code resides, but also other machines that may be available. In this case I will be using my own personal machine to help with the evolving of a program if the need arises, effectively doubling the amount of client consoles that can be used.

There are a few settings one needs to be mindful of when developing learning cases for CGP-IP. The first thing that needs to be decided is what kind of OpenCV based segmentation algorithm is going to be trained.

During this semester, a several datasets using various items were used. These datasets will be explained in detail later. In CGP-IP, two types of images are needed to evolve a program. The first type of image is an input image. It can be in the form of a png, jpg, bmp or any extension that OpenCV recognises. The input image is an example of the environment that
the program will be used in. This environment will vary depending on the scope of the problem. The other type of image is the target image. This image is the expected output of the completed algorithm and is a binary mask (i.e. containing only black or white pixels). Each individual in the program will use the input image as input and the target image will be used to test the output of the evolved algorithm.

In the same folder as the CGP-IP program, two initialisation files need to be present with the various parameters that the program can use. These files have the extension .ini. These files consist of all the different parameters that can be changed by the user when using CGP-IP. There are two of these files that need to be present in the same directory as the executable file, parameters.ini, which contains the name of another .ini file that contains all of the parameters. Figure 1 shows an example of the second of file. It is set up this way so that the code can be run without the need to rebuild the CGP-IP program every time a new dataset needs to be used. It can also be used in such a way that multiple .ini files for different filters can be present in the directory and only the parameters.ini file needs to be changed. For simplicity, the initialisation file that holds all of the parameters will be called ch.ini and the file that holds the pointer to ch.ini will be called parameters.ini.

The target image does not actually have to be a binary image. It can also be the input image with red painted over the part of the image that needs to be segmented. The parameter FilterTargetsForRed, when checked will scan the target image for the colour red (R<=200, G<=50, B<=50) and turn it into a binary image based on whether each pixel contains those shades of red. The red parts of each target image are the areas that contain the subject that the filter will look for.

If for some reason the image has objects that would normally appear in a filter that is being evolved, consistently comes up as a false positive or any other kind of reason, an ignore mask can be made by painting the colour blue (R<=50, G<=50, B<=200) over the parts that need to be ignored. The FilterTargetsForBlue parameter also needs to be set to true in ch.ini.
There are two input directories and three output directories that are used in the initialisation file. The two input folders are used for the input images and target images and the three output folders are for the best individual (usually referred to as the server individual) the predicted output of the server individual and the output, or trace, file. The trace file contains the filter output of every 50th individual, the graph of the genotype of the individual that any viz client evaluates and also the inputs and nodes of the most recently evaluated individual by the viz client. This folder is used to check the outputs of the program and will only be populated by the viz clients.

Another important parameter is the ServerTestCasesCount parameter that is used to keep track of the number of test cases. A test case is an input/target image pair found in the input and target directories. If the amount of image pairs evaluated in a client console does...
not equal this number then the server will reject any individual from that client. The reason that this parameter is important is because if the number of images in the inputs/targets folder is not the same as the number in ServerTestCasesCount, it is assumed that there are images in the test set that are not within the scope. For example, if a client set up for a certain dataset is run with a server for another dataset then the output filter would turn out wrong and therefore be useless.

4.1.1 Datasets

The datasets used in CGP-IP are sets of images that have one or more household object in them. There are at least four images in each set. The first few datasets were used to see if a useable filter could be evolved using CGP-IP. After that, a dataset from an online source was used. The online database was used as a way to compare results from this experiment with results from other experiments to make sure these results were worthwhile. The dataset used was the Weizmann segmentation database.\footnote{This dataset contains images from various royalty free image databases and has three human segmented targets.}

4.1.1.1 Cow Cup

The first dataset used was a coffee cup with a dairy cow pattern printed on it. The images used in this dataset consisted of 7 images of the cup in various orientations and distances from the camera.

This dataset was used twice. The first time, the mask that was used as the target output (henceforth called target) was as accurate as possible. The Microsoft paint tool was used to make the mask as close to pixel perfect as possible. The second time, the mask was just a rectangle that covered the entire cup. The rectangle also covered parts of the image that were not a part of the cup.
4.1.1.2 Container
The second dataset used was a cylindrical metal container. The four images were of the container with different backgrounds. The backgrounds had different lighting and the camera was also a different distance away for each photo.

This dataset was also used twice, using as close to a pixel perfect as possible mask for the first time and a more general square as the mask for the second time.

4.1.1.3 Tea Box
The third dataset used was a simulated box of tea. Each image consisted of objects created to give a diverse set of shapes and sized in one photo. The main focus of these images was a box of tea as mentioned before. The tea box was in various orientations as well as different sizes simulating distances from the ‘camera.’

This dataset was also used twice; however the two uses were not the same as the previous two. The first time, a random set of five images were chosen from a larger set of thirty-three. There was no relation between the orientation of the tea box in one image to the orientation in another. The second time, images that showed the tea box in similar orientations and sizes were chosen.

4.1.1.4 Rep/Rot
The fourth dataset consists of two sets of images of several household objects. In the first set of images, the objects do not move and the photos are taken from different angles. There are four images in each set. The second set of images were all taken from the same angle, but each image has the objects moved around.

Each of the two image sets in this data set will be run once for each object in the images. There are five objects in each image so this data set was used ten times.
4.1.1.5 Weizmann Segmentation Database

The fifth dataset is the Weizmann Segmentation Database. This database consists of several images taken from royalty free image databases. Each image set has a greyscale and colour input image and three human segmented images. The mask in the segmented images is drawn on in red so this will need to be taken into consideration when setting up the parameter files for these images. The images being used from this dataset are b4nature_animals_land009, bream_in_basin, broom07, caterpillar, chain98, and cheeky_penguin.

4.1.2 Main Class, Program.cs

This is the main class of the program. This Class' main function is what gets called when the program runs. The main program is used to set up the code to run in the way it needs to. Because the code is used for research, it is constantly changing. Functions get added and different test setups are made. If a different test setup needs to be used, it is called in the main(args). For this project, the main function only runs Go(args), with the same arguments that it takes. It also has functions to set up servers and clients and also set up the fitness function evaluation.

4.1.2.1 Program.cs function Go(string[] args)

Go() is the largest function in the Program.cs class. It sets up and runs the clients and server depending on the inputs given in args. The inputs are as follows; server, client, viz, consts, spawn and log. These inputs can be typed in any order as the program loops through all of them and sets the relevant variables according to the inputs.

Using client as an input makes a client console. Using viz after client will make the client a visualisation client, printing information about the process as it runs. Using consts after client will make the program only ever evolve parameters that are a part of the parameters class. Using log after client will make that client create and save a log file that will write anything that appears on the console.
Using server as an input will create a Server Console. This console is used to keep track of the progress of all of the clients that are created to talk to it. A server can be made to only start itself or it can be made to create clients by using spawn as an argument after server when running the program.

Once the setup of the server/client is finished, the client functionality is executed. If the console was a server, its functionality is defined in StartServer(). If the client has successfully been set up it will start to generate populations and populating those populations with individuals. If the server has an individual (the best individual so far) the client will use that individual in all populations.

After the client has been populated it goes into a loop that tests the individuals in each population against the fitness function, returning a fitness value. If the fitness for the current individual is better than the fitness of the best individual, (stored on the server) it will replace the old best individual with the new best one. It also periodically checks for any new inputs and targets and will add them to the batch.

If the parameter to evolve population parameters is set to true, the client will attempt to mutate the parameters for the populations. The client will then mutate the individuals in each population by way of crossover.

Once per loop, a signal will be sent to the server console to process any events it needs to. This is usually accepting the new best individual and printing that information to the console.

4.1.2.2 Program.cs function StartServer(bool Spawn)

This is a simple program that creates and manages a server console. It sets up a communications service that clients can connect to and if spawn is set to true it will also generate new consoles that are set up as clients. The number of spawned clients is dependent on the number of processors of the machine being used. The first of the n clients is always a visualisation client. This client will be used to write the information about the individuals to files so they can be viewed at a later time.
4.1.2.3 Program.cs function StartClient()
The StartClient() function is used to set up communication with the server. If the client successfully connects to the server it will return the connection, if it fails it will return null. If the client returns null, it will not attempt any of the client based functionality in Go().

4.1.2.4 Fitness Function Class FitnessFunction.cs
This class is used to find the fitness of an individual. It includes all of the testing needed to return an accurate fitness based of the MCC function (See below).

4.1.2.5 Program.cs function TestPopulation(FitnessFunction FitFunc, CGPPopulation Pop)
This function evaluates an individual and determines its fitness. The function starts by decoding the genotype of the given individual. Once decoded, it evaluates the functions to get a phenotype which is executed to produce an output image. This output is then checked against the target. This is done by checking whether each pixel of the output is the same as the corresponding pixel in the target. The output of the check is in the form of the number of

- True Positives (the target item was there and the predicted output said item was there),
- True Negatives (the target item was not there and the predicted output said item was not there),
- False Positives (the target item was not there and the predicted output said item was there) and
- False Negatives (the target item was there and the predicted output said item was not there).

The resulting output.error is calculated by running the output through the MCC function.
4.1.3 Matthews Correlation Coefficient (MCC) and Fitness Function

The fitness function of CGP-IP needs to be able to accurately determine the fitness while being able to work for any test case given. This use of CGP-IP is mainly focused on binary classification where the outputs are in the form of a picture with each pixel being true (white) or false (black) the function needs to not only keep track of how many pixels were correctly identified when compared to the target image but also whether the predicted output pixels were true when they were meant to be false and vice versa. The fitness function makes a ‘confusion matrix’ which has a size of 2×2 and for every pixel in the output image, populates it with one of four evaluations. These are True Positives, (TP) False Positives, (FP) True Negatives, (TN) and True Negatives (TN) TP is given when the predicted image gave true and the target gave true, TN is given when the predicted image gave negative and the target gave negative, FP is given when the predicted image gave positive and the target gave negative and FN is given when the predicted image gave negative and the target gave positive. These four outputs of the confusion matrix are used to find the Matthews Correlation Coefficient (MCC) of the image. The MCC of an image is calculated as follows.

\[ MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \]

The MCC will have outputs between -1 and 1 where 1 means the output has perfect fitness (all values are TP or TN), 0 means the output is essentially working at random and -1 means the output is perfect but is inverted (All values are FP or FN. This means every pixel was not what it was supposed to be and therefore is the same as all values being TP or TN but all black pixels are white and all white pixels are black).

The fitness of an individual is calculated as follows.

\[ Fitness = 1 - |MCC| \]

With fitness values closer to 0 being more fit.
4.1.4 Program Flow

Below is the flow chart for the CGP-IP program. It has two main sections, the original CGP-IP functionality and the new FilterMaker functionality.

![Flow chart of CGP-IP](image-url)
EVALUATE POPULATIONS

DETERMINE BEST INDIVIDUAL

IS CURRENT INDIVIDUAL THE BEST?

SEND INDIVIDUAL TO SERVER

SAVE SERVER INDIVIDUAL

SET ALL POPULATIONS TO CONTAIN ONLY BEST INDIVIDUAL

MUTATE ALL POPULATIONS

REPEAT FOREVER
The program starts when it is called from the command line. The program can be called with any of three primary arguments and four secondary arguments. The primary arguments are server, client and filter. The secondary arguments are viz, consts spawn and log. Before the main method is called setup is performed, setting program parameters using the parameters file.

Main is only used as a way to determine which of the two program types will be run. If the code is run as a filter maker, it calls that function and if the code is run as CGP-IP, it calls go(). Main also sets up a debugger if necessary.

4.1.4.1 CGP-IP functionality
The main functionality of the CGP-IP code is CGP-IP. This is the process that employs Cartesian Genetic Programming on a number of input images, attempting to evolve a filter of a certain item. This part of the program has two modes, server and client. The server is the central part of the process. All client processes are connected to the server and each best individual is sent to and stored in the server. The server itself doesn't do any processing and will only act when a client connects or sends an individual. The client services perform all of the processing, creating individuals, evaluating the individuals, sending best individuals to the server and mutating individuals.

The CGP-IP process starts with determining the role of the current process. As mentioned above, the process can either be a server or a client. There are also four other arguments that can be used; viz, log, spawn and consts. Using viz will cause the client process to save a visual representation of all best individuals over the course of the evolution. It will also create a window that will display the results of the best individual so far, updating whenever a better individual is found. Log will simply save the output of the client console to a text file. The server does not need to use log as it automatically saves its output to a log file. Spawn is a server specific argument. When a server is started it can be used to create a number of client processes as well. Spawn is the flag used to do so. Lastly, consts is a flag used to determine what attributes of an individual will be mutated. If consts is used, the function and inputs will not be changed. If consts is not used then the function and inputs will be changed. Other individual attributes (known as genes) that can be changes are node
height, node width, feature config and gray value (The gene is actually called gray value instead of grey value). These genes are used as inputs to the various functions, along with connections.

Once the role of the current process is determined, the duties of the role are set up. If the process is a server, a communications service host is set up. If spawn was used as an input argument, a number of new processes are made in new terminal windows as clients. The first of these new clients will be called with viz and log as additional arguments. The server then waits for a connection or an individual. If the process is to be a client, a communications service is created and connected to the server.

From here on in is the functionality of the client. The server process never reaches this point.

The first thing that the client does is create a set of populations of individuals. By default there will be one population with five individuals per population. Each individual is created by creating a graph of 50 randomly created nodes. Each node has a function, three connections, a gray value, height, width, a number of times it can be used in evaluation and a feature configuration. Each of these values are randomly selected when each node is created. If there is an individual to be loaded from an XML file then only that individual will be created. Extra individuals will not be created for a loaded individual until the mutation process.

The next step is the evaluation and mutation process. This process is run until the user stops the client.

The client's populations are evaluated. This process has three main parts, running through the individual using the graph runner and evaluating each node, thresholding the individual's output and determining the fitness of the thresholded output. The running of the node is pretty simple; for each node in the individual's graph; get node's function, get node's inputs, evaluate function using node inputs and node parameters and store result image. Once the graph has been run through, the output image (the output of the last evaluated node) is thresholded using the threshold value stored in the individual.
The fitness of an individual is determined by comparing it to the target and using that comparison as the basis of an MCC function. The comparison is done by checking each pixel against the target. If the output pixel is set to true and the target pixel is also true the True Positive (TP) count is increased by one. If the output pixel is false and the target pixel is true, the False Negative (FN) count is increased by one. If the output pixel is true and the target pixel is false, the False Positive (FP) count is increased by one. Lastly, if the output pixel is false and the target pixel is false, the True Negative (TN) count is increased by one.

The fitness of an individual is defined as $1 - |\text{MCC}|$. A fitness of 0 is the best possible fitness. The MCC of an individual is defined as

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

where TP, TN, FP and FN are defined as above. The MCC function will always return a value $-1 \leq \text{MCC} \leq 1$. An MCC value of 1 means that every pixel was evaluated as TP or TN, meaning that the output image is exactly the same as the target image. An MCC of -1 means that every pixel was evaluated as FN or FP, meaning the output image is the same as the inverse of the target image. An MCC approaching 0 means that the output image is nothing like the target image, having no similarities. Since and MCC of 1 and -1 are essentially the same (a filter finding the target and a filter finding everything except the target are functionally the same) the absolute value of the MCC is used. If the fitness of the current individual is better than the fitness of the server individual or the server does not have an individual stored, the client sends the individual to the server to be used as the new best individual.

Once the populations are evaluated, they need to be mutated. The mutation process sets all individuals in each population to be the same as the server individual and then mutates all individuals in the population except one. The mutation process will randomly change a set number of nodes. The number of nodes changed is equal to the number of nodes in the graph. Since the nodes changed are random, a single node can be mutated several times. Mutation is the last thing done before repeating.
4.1.4.2 FilterMaker Functionality

The FilterMaker is used to create the filter described by an individual. It can take two additional input arguments; img and camera.

The FilterMaker needs an individual to evaluate. This is done by loading an XML file containing the individual. This XML file must be located in the working directory of the CGP-IP executable. Once the individual has been loaded in, a graph runner is initialised. The graph runner then runs through the graph once and makes a list of the evaluated nodes. The reason a list is created here is because only one individual will be used. This means processing overhead can be reduced, even slightly by only checking for evaluated nodes once.

If the argument “img” was used, the FilterMaker will load inputs based on a filepath in the code. Using these inputs, the individual is run once with RunFast. RunFast does the same thing as the original graph running function but it has a lot of reporting functionality removed. It also uses the list of evaluated nodes to loop through the graph instead of checking each node to see if it is evaluated. This is to remove any extra processing time. RunFast also returns the output of the final node. This output is the binary mask that the filter produces. This output image is saved and printed to the screen.

The single image output is used to show the change to filter accuracy when slow functions are removed from the individual.

If “camera” is used as an argument, the output of the filter is only printed to the screen. This process is repeated indefinitely, creating a video of a filtered version of what the camera sees.

The inputs used in this case are single frames from a web cam. They are stored just the same as when “img” is used. The fast runner will run and output an image which will have a threshold applied. The thresholded image will then be printed to the screen.
4.1.5 Local Minimum

A local minimum is a problem that has been known to happen when running CGP-IP. A local minimum is a situation where an evolving filter will get to a point where it will not progress any further. However, the fitness may be relatively high or known to be able to go lower. There are many combinations of functions that can cause this phenomenon but it is not entirely known what causes them, or why with the randomness of mutation they cannot break out of these minima.
7. Progress

5.1 Semester One

The aim of this project is to better understand the processes happening in CGP-IP. Which nodes containing which functions are used and how often, whether the same learning scenario run multiple times will tend to a similar solution and whether different but similar scoped learning scenarios will tend to a similar solution.

For this semester, the aim was to transfer the code from the old uses and get it to work on the new machine. This involved combing through the program and fixing any problems with the code. On the first run, an out of bounds exception was thrown. On further exception, it was found that the main function was calling a test() function that was looking for files that did not exist. The test functions were not needed so the call to that function call was removed.

The next error was another null pointer error caused by the directory of the input and target image folder not existing. The reason for this was because the directory was not set in the code but by the initialisation file. At the time of finding this error, I did not know about the initialisation files and it was not until there was a meeting with the supervisor that the initialisation files were present.

After the null pointer exceptions were fixed the code compiled and run. Due to the nature of the code, time is needed to develop the segmentation so it was left to run for a day. The output of the client consoles showed that only one number was changing. Upon inspection of the code it was found that the number was the evaluation count. One of the outputs of the program was the current individual being evaluated (this was not discovered until sometime after) with the lack of knowledge of the outputs of the client it was impossible to tell whether the filter was evolving just by looking at the individual, so once the code ran for the one day the outputs were shown to the supervisor who explained what TP, FP, TN and FN meant. The individual that was shown to the supervisor was shown to have TN for all pixels of the predicted. At this time the code inspection was put on hold while the supervisor set up a work station that could run the code and this report could be written.
While looking for the fix to the problem before the meeting with the supervisor a problem with the ServerTestCasesCount was fixed.

After the workstation was set up, the code was run and inspected with the help of the supervisor. As the Go() function was stepped through the supervisor explained what each function attempted to do. It was in the FitnessFunction.cs file that the problem finally came to light. In the function TestCase.Compare(<inputs>) it was found that the test case was a completely black square. There must have been something wrong with the initialisation of TestCase so the class containing its creation was inspected next. It was here that a parameter FilterRedForPositive and FilterBlueForIgnore was found. As explained earlier in the report, FilterRedForPositive is used as an alternative to using a binary mask. The entire image, with no part being the correct shade of red, was completely set to black causing the learning to be incorrect and the program not evolving. The red in the input image had a blue value of 51 and therefore was not being picked up by the filter. Once these values were set in the initialisation file, the program successfully started evolving the filter as intended.

The kind of program that was trying to be evolved was one that scanned Figure 2 and not find anything in the picture as the expected output was an image with all black pixels. According to the MCC function, all TN values will give a value of 0 which corresponds to a fitness of 0 so CGP-IP was working properly from the start. It was only human error that was stopping it from giving the output what was actually wanted.

Once the filtering parameters were corrected, CGP-IP was able to work the way it was intended; to make a filter that only found the colour orange. The type of filter and also the small size of the input images meant that the filter was evolved to a fitness of under 0.025 in less than 5 minutes. Of the 15376 pixels there were 23 False Positive pixels and 55 False Negative pixels. This error can be attributed to the fact that the input and target images are scaled down to 124×124 which leads to pixels on edges of the circles to be melded together to create a colour not in the original image.
5.2 Semester Two

5.2.1 Translating CGP-IP into the Linux Platform

At the beginning of the semester, it was decided that some time would be used to attempt to migrate the program from the windows platform to the Linux platform. This was because of the HPC facilities that QUT has. The HPC centre* is a large distributed computing system that has a very large amount of resources than can greatly speed up the process of evolving the filters that were needed in this project. The plan was to use the HPC centre instead of the home computer to do all, or at least most, of the processing thus, lowering the time needed to evolve a useable filter. Due to time restraints however, the HPC was not utilised.

Running the program on a Linux machine with no changes worked up until the program attempted to load an image. It was never assumed that the program would work right off the bat as it was developed in Microsoft Visual Studio and that IDE has known compatibility problems even with machines running windows. Running the code was used as a way to figure out when to begin.

The cause of the problem was with the .net OpenCV wrapper emguCV. According to the help section of the emguCV website, the error being thrown was due to the .dll files not being of the appropriate type. Instructions on how to install the emguCV dll files were followed and the new files were added to the project in place of the ones used in the windows version. Unfortunately, this fix did not solve the problem.

Several attempts to make the program resulted in failure each time. Installing a different version of the dll files resulted in files that could not be found, moving all of the .cs files to a completely fresh project resulted in the same error that was encountered after installing the first set of dll files. After three weeks of not making any progress, the conversion was dropped as any more time would hinder the progress of the actual project too much.

During the attempts to convert over to a Linux environment, some datasets were created and used to attempt to develop a filter for an object.

Translating CGP-IP from a windows architecture to a Linux architecture is mainly figuring out which parts of the .NET framework haven’t been implemented and using the correct version
of files. The first big hurdle was installing emguCV on the Linux machine. The installation instructions on the emguCV website did not say anything about versioning conflicts.

Firstly, the latest version of emguCV would not work on Ubuntu 14.04 which is what was installed on the machine. The installation instructions did mention this and advised to checkout an earlier version of emguCV, version 2.4.9.

One of the steps in the installation instructions is to install Mono, a .NET package. However the instructions did not mention that the version of Mono at the time of the instruction’s creation was a different version than the one currently available. The terminal commands given would install the latest version of Mono, version 4.0.4. This version of Mono was not able to be used in the build process of the checked out version of emguCV as the files that emguCV was dependant on were not the same in Mono 4.0.4. It was not obvious that this was the case and a lot of troubleshooting was done before the answer to the problem was found.

The Mono.NET package is an open source implementation of Microsoft’s .NET framework, but it is not a complete implementation. There are several aspects of .NET that Mono does not have implemented, one of which is used in the CGP-IP program. The data visualisation library charting capabilities are not implemented. The charts are used to give a visual representation of the progress that is being made. This is not imperative to the main processes so it was able to be disabled without causing any problems.

The final problem found when translating the code was a discrepancy with how integer overflow was treated between Microsoft.NET and Mono.NET. The MCC calculation function is used to calculate the MCC of the individual based on the comparison with the ideal final filter. Its equation, as stated earlier is:

$$ MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} $$

The denominator of this function is prone to blowing out of proportion with even relatively small images. The dataset that was being used to test the functionality of the Linux based code was the RepPaper dataset. This dataset contains 4 images that are 780*585 pixels. Several runs of the code had TN, TP, FN, and FP values such that when multiplied together,
left the values inside the square root to be larger than the maximum value of a 64 bit integer. This meant that the integer values were overflowing and returning the wrong values which lead to very wrong fitness values.

```csharp
if (((TP + FP == 0) || (TP + FN == 0) || (TN + FP == 0)) || (TN + FN == 0))
    return 0;

Int64 xbigTP = ((long)(TP / Parameters.TPWeight));
Int64 xbigFP = ((long)(FP / Parameters.FPWeight));
Int64 xbigTN = ((long)(TN / Parameters.TNWeight));
Int64 xbigFN = ((long)(FN / Parameters.FNWeight));

Int64 xn = (xbigTP * xbigTN) - (xbigFP * xbigFN);
Int64 xd = (xbigTP + xbigFP) * (xbigTP + xbigFN) * (xbigTN + xbigFP) * (xbigTN + xbigFN);
Int64 sqsd = (Int64)Math.Sqrt(xd);
double xd1 = (xbigTP + xbigFP) * (xbigTP + xbigFN);
double xd2 = (xbigTN + xbigFP) * (xbigTN + xbigFN);
double sqsd1 = Math.Sqrt(xd1) * Math.Sqrt(xd2);

double xndouble = Convert.ToDouble(xn);
if (Parameters.Visualize)
    Reporting.Show("MCC is \>_1");
```

**Figure 4:** Picture of IDE showing the correct and incorrect MCC values
As shown in figure(x), using the overflowed integer, xd results in xsmallMcc2 not in the bounds of \(-1 \leq MCC \leq 1\). While splitting xd into two parts yields a better result, xsmallMcc.

The overflow problem was remedied by splitting up the denominator into two sections and then combining them once they were reduced in size by the square root. This operation follows the rule

$$\sqrt{a \times b} = \sqrt{a} \times \sqrt{b}$$

Once all of these problems were resolved, the code ran without any errors.

5.2.2 Re-Purposing the Graph Runner to Create a Filter

The main difficulty with this task was to alter the already existing code to execute its secondary use without breaking its primary use. While a lot of the existing code was very useful in implementing the filter, there was still a lot missing.

Firstly, the graph runner that would run through and evaluate an individual did not return anything. Instead it would save the input images to the requested filepath. It was important for the runner to return an image, or at least store it so that it could be used. Saving and loading images would not be good enough to use with a webcam. It was decided that the current runner function should not be altered too much and so, a second runner function was created.

The new function was called RunFast because it was essentially the same as RunGraph but with any visualisation code removed. RunFast took the individual’s graph, two input images, a list of evaluated nodes and a tag. The individual’s graph was the graph of nodes that held the image operations, their connections and other important information. Because of the amount of information needed for each node, the XML file output of the primary code needed to be used to make sure any variables weren’t missed. There was already functionality to load these XML files so no extra code was created.

The two input images were a grayscale version of the original input image and the first image normalised with Gaussian smoothing. These input images by default were taken from a webcam, but the code was also altered to also use saved image files for testing purposes.
The list of evaluated nodes had to be calculated before the runner could be executed. When the individual was loaded from the XML file, a function to check for evaluated nodes was run. Its output was the list of evaluated nodes to be used in the RunFast function.

In the primary functionality, the tag input was used as the name of the image at each evaluated node. The primary functionality would save each of these images whereas the secondary functionality did not. The tag input is used only because without it, many of the functions in the GraphRunner class would not work.

At each evaluated node, the runner would check which operation it needed to perform on its input connections. Each node was connected to either another node or one of the two input images. Each node’s output image was stored and then used when needed for later nodes that used it as a connection. The output image, which is the image stored in the last node was returned once the entire graph had been run.

Once the output image was returned, it needed to be thresholded. The threshold limit was also in the XML file and therefore loaded in when the individual was. The final, thresholded image was then printed to the screen in a window. If the webcam was used, this process would be repeated until the program was closed. If a saved image was used, the program would end by itself.
8. Experiments

6.1 Experiment One
CGP-IP is a process of evolving a filter for a specific object in an image using an input image, a target image and a set of randomly generated functions from a pre-defined set. Due to the random nature of evolution, it is described as a stochastic process. But are there any similarities between different filters evolved from the same dataset. This is the goal of the second experiment.

To complete the experiment, several filters were made using several datasets. Filters were also evolved on two separate computers with no communication between each other. There were at least four filters developed per dataset per computer.

Each filter was analysed and various attributes were recorded. These attributes were types of functions, number of functions, filter output, fitness and time taken to reach fitness.

6.2 Experiment Two
The second experiment was to test the success of the code migration to a Linux based OS. As discussed in the Methodology section, there were several hurdles that needed to be passed before the code did its job without anything wrong happening. But it’s not enough to just say that the code worked properly. It needs to be tested.

For this test, several filters were evolved using different datasets on a computer with a Linux OS running version 14.04. Each filter was evolved for a similar length of time as an equivalent filter from a Windows OS.

Each filter was analysed and various attributes were recorded. These attributes were types of functions, number of functions, filter output, fitness and time taken to reach fitness. The attributes were then compared with each other and from that a decision was made as to whether the filters were similar enough to be deemed correct.

6.3 Experiment Three
The third experiment was to test the limits of the filter making process using one individual. It was discovered during development that some filters were very slow to evaluate. They would take more than half a second to complete the operation. For a single image input,
this result is satisfactory. Attempting to find an object in a large image, or using the filter on low FPS security camera footage would be okay with the slow filters. However, other applications such as high fps surveillance or robots completing tasks in real time would be hindered by the high processing time that is currently experienced with some filters. It was found that the reason for the low evaluation times was due to a select few operations that would take up the majority of the processing time. (>400ms)

The goal of this experiment is to find as many functions as possible that hinder performance time. When found, the function would be omitted from the runner by setting the output to equal the input for that function. The execution time if the “fast” filter would be recorded and compared to the “slow” filter. The fast and slow filters’ outputs would also be compared, checking for any change in accuracy.
9. Experiment Results

7.1 Experiment One

7.1.1 Container Dataset
The container dataset was run five times and produced five different filters. They are as follows

7.1.1.1 Filter One
Fitness: 0.317292214091877

Nodes used: 6

Functions: ADAPTIVETHRESHOLD, SHIFTRIGHT, SMOOTH, SMOOTHBILATRAL, ABSTHRESHOLD, SMOOTHBILATRAL.

Best Ind:

Time taken: 2 hours 30 minutes
7.1.1.2 Filter Two

Fitness: 0.254300979469829

Nodes used: 10

Functions: AVG, BLACKHAT, DILATE, ADD3, ABSTHRESHOLD, Rt, SHIFTUP, TOPHAT, SMOOTH, TOPHAT

Best Ind:

Figure 6: Filter Two best individual

Time Taken: 11 hours
7.1.1.3 Filter Three

Fitness: 0.206962103510795

Nodes used: 9

Functions: LAPLACE, MAX, BLACKHAT, RESIZE, CLOSE, SHIFTUP, RESAMPLE, DILATE, ABSTHRESHOLD.

Best Ind:

Figure 7: Filter Three best individual

Time Taken: 10 hours
7.1.1.4 Filter Four

Fitness: 0.0991310036753668

Nodes used: 10

Functions: MAXConst, SUBTRACTConst, GRADIENT, COMP2, ADD3, EDGEMAG, SUBTRACT, Rq, SHIFTDOWN, OPEN

Best Ind:

Figure 8: Filter Four best individual

Time Taken: 3 hours
7.1.1.5 Filter Five

Fitness: 0.28477891122366067

Nodes used: 8

Functions: UNSHARPEN, UNSHARPEN, TOPHAT, RESIZE, ABSDIFF, EDGEMAG, SMOOTH, SMOOTH

Best Ind:

Figure 9: Filter Five best individual

Time Taken: 4 hours
Table 1: Container Function Count

<table>
<thead>
<tr>
<th>Function</th>
<th>Count</th>
<th>Appears in…</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNSHARPEN</td>
<td>2</td>
<td>5, 5,</td>
</tr>
<tr>
<td>TOPHAT</td>
<td>3</td>
<td>5, 2, 1,</td>
</tr>
<tr>
<td>RESIZE</td>
<td>2</td>
<td>5, 3,</td>
</tr>
<tr>
<td>ABSDIFF</td>
<td>1</td>
<td>5,</td>
</tr>
<tr>
<td>EDGEMAG</td>
<td>2</td>
<td>5, 4,</td>
</tr>
<tr>
<td>SMOOTH</td>
<td>4</td>
<td>5, 5, 2, 1,</td>
</tr>
<tr>
<td>MAXConst</td>
<td>1</td>
<td>4,</td>
</tr>
<tr>
<td>SUBTRACTConst</td>
<td>1</td>
<td>4,</td>
</tr>
<tr>
<td>GRADIENT</td>
<td>1</td>
<td>4,</td>
</tr>
<tr>
<td>COMP2</td>
<td>1</td>
<td>4,</td>
</tr>
<tr>
<td>ADD3</td>
<td>2</td>
<td>4, 2,</td>
</tr>
<tr>
<td>SHIFTDOWN</td>
<td>1</td>
<td>4,</td>
</tr>
<tr>
<td>SUBTRACT</td>
<td>1</td>
<td>4,</td>
</tr>
<tr>
<td>Rq</td>
<td>1</td>
<td>4,</td>
</tr>
<tr>
<td>OPEN</td>
<td>1</td>
<td>4,</td>
</tr>
<tr>
<td>LAPLACE</td>
<td>1</td>
<td>3,</td>
</tr>
<tr>
<td>MAX</td>
<td>1</td>
<td>3,</td>
</tr>
<tr>
<td>BLACKHAT</td>
<td>2</td>
<td>3, 2,</td>
</tr>
<tr>
<td>CLOSE</td>
<td>1</td>
<td>3,</td>
</tr>
<tr>
<td>SHIFTUP</td>
<td>2</td>
<td>3, 2,</td>
</tr>
<tr>
<td>RESAMPLE</td>
<td>1</td>
<td>3,</td>
</tr>
<tr>
<td>DILATE</td>
<td>2</td>
<td>3, 2,</td>
</tr>
<tr>
<td>ABSTHRESHOLD</td>
<td>3</td>
<td>3, 2, 1,</td>
</tr>
<tr>
<td>AVG</td>
<td>1</td>
<td>2,</td>
</tr>
<tr>
<td>Rt</td>
<td>1</td>
<td>2,</td>
</tr>
<tr>
<td>ADAPTIVETHRESHOLD</td>
<td>1</td>
<td>1,</td>
</tr>
<tr>
<td>SHIFTRIGHT</td>
<td>1</td>
<td>1,</td>
</tr>
<tr>
<td>SMOOTHBILATRAL</td>
<td>2</td>
<td>1, 1,</td>
</tr>
</tbody>
</table>

Tophat, smooth, and absthreshold are all used more than two times in the five filters. This would make it seem that those three functions would be important to a good filter for the container. Filter four, the best filter does not use any of these functions though. Filter four is also the most unique filter, only sharing a function with two other filters.
As shown in figures(x1, x2, x3, x5), the hardest part of this filter’s learning process is finding the entire container in the first image. This is because the lighting of the first image is different to the lighting in the second image.

7.1.2 Cow Cup Dataset

7.1.2.1 Filter One

Fitness: 0.399330946544577

Nodes used: 17

Functions: EDGEMAG, MAX, SUBTRACT, THRESHOLD, QUANTIZE, SMOOTHGAUSSIAN, SUBTRACT, EDGEMAG, ADD, LMAJ, LMAJ, SUBTRACT, AND, AVG, SHIFT, DILATE, CLOSE

Best Ind:

Figure 10: Filter One best individual

Time Taken: 18 hours
7.1.2.2 Filter Two

Fitness: 0.381823740926714

Nodes used: 10

Functions: AVG, LAPLACE, ADAPTIVETHRESHOLD, ABSDIFF, GRADIENT, ABSDIFF, AND, Rv

Best Ind:

Figure 11: Filter Two best individual

Time Taken: 6 hours 30 minutes
7.1.2.43 Filter Three

Fitness: 0.594229473716161

Nodes used: 7

Functions: Rz, SMOOTHBILATRAL, MINConst, ADD, BLACKHAT, MULTIPLY, QUANTIZE

Best Ind:

Figure 12: Filter Three best individual

Time Taken: 12 hours
7.1.2.4 Filter Four

Fitness: 0.46982639395185

Nodes used: 9

Functions: OPEN, GRADIENT, ADDConst, SUBTRACT, OPEN, Ra, SMOOTHBILATRAL, MIN, Rq

Best Ind:

Figure 13: Filter Four best individual

Time Taken: 10 hours
<table>
<thead>
<tr>
<th>Function</th>
<th>Count</th>
<th>Appears in...</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDGEMAG</td>
<td>2</td>
<td>1, 1,</td>
</tr>
<tr>
<td>MAX</td>
<td>1</td>
<td>1,</td>
</tr>
<tr>
<td>SUBTRACT</td>
<td>3</td>
<td>1, 1, 4,</td>
</tr>
<tr>
<td>THRESHOLD</td>
<td>1</td>
<td>1,</td>
</tr>
<tr>
<td>QUANTIZE</td>
<td>2</td>
<td>1, 3,</td>
</tr>
<tr>
<td>SMOOTHGAUSSIAN</td>
<td>1</td>
<td>1,</td>
</tr>
<tr>
<td>SUBTRACT</td>
<td>1</td>
<td>1,</td>
</tr>
<tr>
<td>ADD</td>
<td>2</td>
<td>1, 3,</td>
</tr>
<tr>
<td>LMAJ</td>
<td>1</td>
<td>1,</td>
</tr>
<tr>
<td>AND</td>
<td>2</td>
<td>1, 2,</td>
</tr>
<tr>
<td>SHIFT</td>
<td>1</td>
<td>1,</td>
</tr>
<tr>
<td>DILATE</td>
<td>1</td>
<td>1,</td>
</tr>
<tr>
<td>CLOSE</td>
<td>1</td>
<td>1,</td>
</tr>
<tr>
<td>AVG</td>
<td>1</td>
<td>2,</td>
</tr>
<tr>
<td>LAPLACE</td>
<td>1</td>
<td>2,</td>
</tr>
<tr>
<td>ADAPTIVETHRESHOLD</td>
<td>1</td>
<td>2,</td>
</tr>
<tr>
<td>ABSDIFF</td>
<td>1</td>
<td>2,</td>
</tr>
<tr>
<td>GRADIENT</td>
<td>2</td>
<td>2, 4,</td>
</tr>
<tr>
<td>ABSDIFF</td>
<td>1</td>
<td>2,</td>
</tr>
<tr>
<td>Rv</td>
<td>1</td>
<td>2,</td>
</tr>
<tr>
<td>Rz</td>
<td>1</td>
<td>3,</td>
</tr>
<tr>
<td>SMOOTHBILATRAL</td>
<td>2</td>
<td>3, 4,</td>
</tr>
<tr>
<td>MINConst</td>
<td>1</td>
<td>3,</td>
</tr>
<tr>
<td>BLACKHAT</td>
<td>1</td>
<td>3,</td>
</tr>
<tr>
<td>MULTIPLY</td>
<td>1</td>
<td>3,</td>
</tr>
<tr>
<td>OPEN</td>
<td>1</td>
<td>4,</td>
</tr>
<tr>
<td>ADDConst</td>
<td>1</td>
<td>4,</td>
</tr>
<tr>
<td>Ra</td>
<td>1</td>
<td>4,</td>
</tr>
<tr>
<td>MIN</td>
<td>1</td>
<td>4,</td>
</tr>
<tr>
<td>Rq</td>
<td>1</td>
<td>4,</td>
</tr>
</tbody>
</table>
No functions appear more than twice in this dataset. All of the datasets are pretty dissimilar and have only one or two functions in common with other filters. The best filter had a fitness of 0.38 and was evolved in the shortest timeframe, similarly to the first dataset. Though, due to the lack of crossover between filters, it’s hard to say whether the filters were stuck in a local minimum or they just needed time to create a more accurate solution.
7.1.3 Rep/Rot Dataset – Calculator (RepCalc)

This set of filters were evolved from the RepCalc section of the Rep/Rot database.

7.1.3.1 Filter One

Fitness: 0.401521483631597

Nodes used: 16

Functions: COMP, MINConst, SHIFT, RESIZE, ADD, RESIZE, SMOOTH, SHIFT, GRADIENT, UNSHARPEN, UNSHARPEN, ERODE, RESAMPLE, OTSU, OR, SMOOTH

Best Ind:

Figure 14: Filter One best individual

Time Taken: 10 hours
**7.1.3.2 Filter Two**

Fitness: 0.272819384501658

Nodes used: 8

Functions: Rq, GRADIENT, THRESHOLDINV, ABSDIFF, Rp, CLOSE, Rv, SMOOTH

Best Ind:

![Figure 15: Filter Two best individual](image)

Time Taken: 9 hours 30 minutes
7.1.3.3 Filter Three

Fitness: 0.10761772618082

Nodes used: 22

Functions: ADD, SHIFTDOWN, SQRT, Rz, ADD, MIN, THRESHOLD, OPEN, OR, GRADIENT, UNSHARPEN, SHIFT, Rz, SMOOTHMEDIAN, CLOSE, GABOR, CLOSE, EQ, MAX, MULTIPLY, ADD3, SHIFT

Best Ind:

Figure 16: Filter Three best individual

Time Taken: 15 hours 15 minutes
7.1.3.4 Filter Four

Fitness: 0.675788342170004

Nodes used: 12

Functions: ABSTHRESHOLD, SMOOTHBILATRAL, EDGEANGLE, CONST, SMOOTH, UNSHARPEN, SHIFTDOWN, MULTIPLY, DILATE, Rt, SMOOTH, COMP

Best Ind:

Figure 17: Filter Four best individual

Time Taken: 11 hours
7.1.3.5 Filter Five

Fitness: 0.301139751744905

Nodes used: 23

Functions: SHIFTDOWN, TOPHAT, MULTIPLYConst, CONST, SMOOTHBILATRAL, THRESHOLDINV, THRESCOMPHOLDINV, DILATE, SMOOTH, MAX, CANNY, ABSTHRESHOLD, SHIFTDOWN, COMP, EDGEANGLE, LMAJ, COMP, DIVIDE, COMP, NOT, ABSDIFF, ADD3, CLOSE

Best Ind:

Figure 18: Filter Five best individual

Time Taken: 17 hours
<table>
<thead>
<tr>
<th>Function</th>
<th>Count</th>
<th>Appears in…</th>
<th>Function</th>
<th>Count</th>
<th>Appears in…</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP</td>
<td>5</td>
<td>1, 4, 5, 5, 5,</td>
<td>TOPHAT</td>
<td>1</td>
<td>5,</td>
</tr>
<tr>
<td>MINConst</td>
<td>1</td>
<td>1,</td>
<td>MULTIPLYConst</td>
<td>1</td>
<td>5,</td>
</tr>
<tr>
<td>RESIZE</td>
<td>2</td>
<td>1, 1,</td>
<td>THRESCOMPHOLDINV</td>
<td>1</td>
<td>5,</td>
</tr>
<tr>
<td>ADD</td>
<td>3</td>
<td>1, 3, 3,</td>
<td>CANNY</td>
<td>1</td>
<td>5,</td>
</tr>
<tr>
<td>OR</td>
<td>2</td>
<td>1, 3,</td>
<td>LMAJ</td>
<td>1</td>
<td>5,</td>
</tr>
<tr>
<td>SMOOTH</td>
<td>6</td>
<td>1, 1, 2, 4, 4, 5,</td>
<td>NOT</td>
<td>1</td>
<td>5,</td>
</tr>
<tr>
<td>SHIFT</td>
<td>5</td>
<td>1, 1, 3, 3, 3,</td>
<td>DIVIDE</td>
<td>1</td>
<td>5,</td>
</tr>
<tr>
<td>GRADIENT</td>
<td>3</td>
<td>1, 2, 3,</td>
<td>MIN</td>
<td>1</td>
<td>3,</td>
</tr>
<tr>
<td>UNSHARPEN</td>
<td>4</td>
<td>1, 1, 3, 4,</td>
<td>THRESHOLD</td>
<td>1</td>
<td>3,</td>
</tr>
<tr>
<td>ERODE</td>
<td>1</td>
<td>1,</td>
<td>OPEN</td>
<td>1</td>
<td>3,</td>
</tr>
<tr>
<td>RESAMPLE</td>
<td>1</td>
<td>1,</td>
<td>SMOOTHMEDIAN</td>
<td>1</td>
<td>3,</td>
</tr>
<tr>
<td>OTSU</td>
<td>1</td>
<td>1,</td>
<td>GABOR</td>
<td>1</td>
<td>3,</td>
</tr>
<tr>
<td>Rq</td>
<td>1</td>
<td>2,</td>
<td>EQ</td>
<td>1</td>
<td>3,</td>
</tr>
<tr>
<td>THRESHOLDINV</td>
<td>2</td>
<td>2, 5,</td>
<td>MAX</td>
<td>2</td>
<td>3, 5,</td>
</tr>
<tr>
<td>ABSDIFF</td>
<td>2</td>
<td>2, 5,</td>
<td>MULTIPLY</td>
<td>2</td>
<td>3, 4,</td>
</tr>
<tr>
<td>Rp</td>
<td>1</td>
<td>2,</td>
<td>ADD3</td>
<td>2</td>
<td>3, 5,</td>
</tr>
<tr>
<td>CLOSE</td>
<td>4</td>
<td>2, 3, 3, 5,</td>
<td>ABSTHRESHOLD</td>
<td>2</td>
<td>4, 5,</td>
</tr>
<tr>
<td>Rv</td>
<td>1</td>
<td>2,</td>
<td>SMOOTHBILATRAL</td>
<td>2</td>
<td>4, 5,</td>
</tr>
<tr>
<td>SHIFTDOWN</td>
<td>4</td>
<td>3, 4, 5, 5,</td>
<td>EDGEANGLE</td>
<td>2</td>
<td>4, 5,</td>
</tr>
<tr>
<td>SQRT</td>
<td>1</td>
<td>3,</td>
<td>CONST</td>
<td>2</td>
<td>4, 5,</td>
</tr>
<tr>
<td>Rz</td>
<td>2</td>
<td>3, 3,</td>
<td>DILATE</td>
<td>2</td>
<td>4, 5,</td>
</tr>
<tr>
<td>Rt</td>
<td>1</td>
<td>4,</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: RepCalc function count

The filters created form this database are much longer than the others. Two of the filters used more than twenty nodes, with a third using twelve nodes. Many filters used the same function two or even three times. There were also seven functions that were used by at least three filters. Of these filters, four were used by the best filter.

The four functions that were used by the best filter and at least two other filters were SHIFT, GRADIENT, UNSHARPEN, CLOSE and SHIFTDOWN.

Because these functions are not only used by a filter with a good fitness, but also by several other filters they can be seen as important functions to make a filter for the calculator in RepCalc.
7.2 Experiment Two

7.2.1 RepPaper

A filter based off the Rep/Rot database with the paper as the main focus was evolved on a machine running Linux 14.04. The filter was evolved for fifteen hours and ended up with a fitness of 0.089118724. The filter correctly identifies about 80 percent of the paper with very few false positives. The output of the best individual is shown in figure 4. The green overlay shows true positives, the red overlay shows false negatives and the blue overlay shows false positives.
When the target image of this dataset was made. The default red colour in GIMP was used. GIMP has a feature that attempts to smooth changes to an image to better fit in the background. In this case, the smoothing caused some pixels in the target images to not fall under the limit of “red” for CGP-IP. Therefore some false positive pixels can be seen on the paper.

To confirm that the evolved filter would actually work, it was run through the FilterMaker. The output is shown on the left in figure 19 below.

Figure 19: Left, filter output. Right, evolution output
7.2.2 TeaBox

A filter evolved using the TeaBox dataset was evolved on the same Linux based machine as the RepPaper filter was. The filter was evolved for eighteen hours and ended up with a fitness of 0.1649944. The filter correctly identifies about ninety percent of the Teabox, with one input image having a few false positives on other boxes.

To confirm that the evolved filter would actually work, it was run through the FilterMaker. The output is shown on the left in figure 20 below.

Figure 20: Left, filter output. Right, evolution output

![Figure 20: Left, filter output. Right, evolution output](image)

![Figure 21: TeaBox output](image)
7.3 Experiment Three

7.3.1 Filter One

The first filter used was developed using the TeaBox dataset. This dataset had a fitness of 0.24544508291668085 and used 12 out of a possible 50 nodes. The functions used, in order were EDGEANGLE, SMOOTHMEDIAN, SMOOTHBILATRAL, SHIFT, SMOOTHMEDIAN, GRADIENT, SMOOTH, SMOOTHBILATRAL, Rq, DILATE and SMOOTHBILATRAL.

Running the filter without any alterations took approximately 0.72 seconds an excerpt from the log file shown below shows that the SMOOTHBILATERAL function takes a very long time in comparison to the other functions.

12:43:15 PM   EDGEANGLE:  00:00:00.0096238
12:43:15 PM   SMOOTHMEDIAN:  00:00:00.0032683
12:43:15 PM   SMOOTHBILATRAL:  00:00:00.0032105
12:43:15 PM   SHIFT:  00:00:00.0056927
12:43:15 PM   SMOOTHMEDIAN:  00:00:00.0031219
12:43:15 PM   GRADIENT:  00:00:00.0023234
12:43:15 PM   SMOOTH:  00:00:00.0002617
12:43:15 PM   SMOOTHBILATRAL:  00:00:00.3442429
12:43:15 PM   Rq:  00:00:00.0323316
12:43:15 PM   DILATE:  00:00:00.0005351
12:43:15 PM   SMOOTHBILATRAL:  00:00:00.3302906
12:43:15 PM   SUBTRACT:  00:00:00.0001372
12:43:15 PM   -----Evaluated. 00:00:00.7359730 seconds taken---------
Bilateral smoothing works similarly to blur functions. The biggest difference is that bilateral smoothing is edge aware. This means that the smoothing function takes into consideration large differences in nearby pixels. This gives the effect shown below in figure [22].

![Original, Gaussian, Bilateral](image)

Figure 22: Difference between Gaussian and bilateral smoothing. Source: [http://ftparmy.com/64316-bilateral-filter.html](http://ftparmy.com/64316-bilateral-filter.html)

This extra consideration causes the processing time to become much larger than other kinds of filters.

An input/output pair using the filter is shown below in figure [23].

![Input/Output](image)

Figure 23: Left, Slow filter output. Right, TeaBox input

This filter's fitness is approximately 0.2454. It is evident by the white spots that are not in the centre of the image. There is about 50% false positives in the filter but most of the blue teabox is covered by it.
Since the largest amount of processing time is taken by the two SMOOTHBILATERAL functions, they were omitted from the function set. Instead, the output image was set to be the same as the input image.

The FilterMaker was run again with the “fast” filter. The output image and log excerpt is shown below.

12:43:52 PM  EDGEANGLE:  00:00:00.0101453
12:43:52 PM  SMOOTHMEDIAN:  00:00:00.0032598
12:43:52 PM  SMOOTHBILATRAL:  00:00:00.0000078
12:43:52 PM  SHIFT:  00:00:00.0055653
12:43:52 PM  SMOOTHMEDIAN:  00:00:00.0032121
12:43:52 PM  GRADIENT:  00:00:00.0028178
12:43:52 PM  SMOOTH:  00:00:00.0004966
12:43:52 PM  SMOOTHBILATRAL:  00:00:00.0000097
12:43:52 PM  Rq:  00:00:00.0329942
12:43:52 PM  DILATE:  00:00:00.0025840
12:43:52 PM  SMOOTHBILATRAL:  00:00:00.0000086
12:43:52 PM  SUBTRACT:  00:00:00.0001025

12:43:52 PM  -----Evaluated.  00:00:00.0620881 seconds taken--------
The speed of the filter after SMOOTHBILATRAL was removed increased tenfold, taking 0.67 seconds of the evaluation time. The accuracy didn’t change a large amount either. There are a couple of spots that are bigger, but only a little. In this case, removing the SMOOTHBILATRAL function did improve the filter’s speed enough to make the accuracy drop worth it.
7.3.2 Filter Two

The second filter used was one evolved using the Rep/Rot dataset with the focus being the metal serviette holder. This filter had a fitness of 0.085865842324010982 and used ten out of the possible fifty nodes. The functions used in order were TOPHAT, SHIFTDOWN, RESAMPLE, CLOSE, Rv, OPEN, SMOOTHBILATRAL, OTSU, AND and ADDConst.

The filter was run without any alterations and the average time taken per image was approximately 0.38 seconds and in the excerpt of the log shown below the bilateral smoothing is again taking up most of the processing power.

```
2:08:55 PM  TOPHAT:  00:00:00.0406572
2:08:55 PM  SHIFTDOWN:  00:00:00.0019332
2:08:55 PM  RESAMPLE:  00:00:00.0003811
2:08:55 PM  CLOSE:  00:00:00.0169148
2:08:55 PM  Rv:  00:00:00.0187402
2:08:55 PM  OPEN:  00:00:00.0041436
2:08:55 PM  SMOOTHBILATRAL:  00:00:00.2923192
2:08:55 PM  OTSU:  00:00:00.0006384
2:08:55 PM  AND:  00:00:00.0003840
2:08:55 PM  ADDConst:  00:00:00.0003157
2:08:55 PM  ---- Evaluated. 00:00:00.3779921 seconds taken--------
```
This filter has a fitness of about 0.086 its accuracy is very high. The filter is accurately finding the serviette holder with very few wrong pixels. According to the XML file there are only 1500 of the ~440000 pixels that are wrong.

FilterMaker was run for a second time using omitting the SMOTTHBILATRAL function, the same as with filter one. This omission also greatly reduced the processing time, bring it down 0.29 seconds to 0.08 seconds.
The filter speed has been reduced to be less than one one hundredth of a second and the accuracy has not noticeably changed. The change in fitness between these two images would be very small. Much smaller than the change in fitness of the first filter. The amount of accuracy that the bilateral filter adds to this filter is heavily outweighed by the slowness of its evaluation speed.
7.3.3 Filter Three
The third filter was one evolved using the Rep/Rot dataset, focusing on the Calculator. It had a fitness of 0.272819384501658 and used eight of the possible fifty nodes. The functions used, in order were Rq, GRADIENT, THRESHOLDINV, ABSDIFF, Rp, CLOSE, Rv and SMOOTH.

The filter was run without any alterations and the average time taken for a frame to evaluate was 0.23 seconds. In the excerpt shown below, the main reason for the slow evaluation is the Gradient function.

![Image](image.png)

**Figure 27**: Left, Slow filter output. Right, RepCalc input

2:47:37 PM  Rq: 00:00:00.0216118
2:47:37 PM  GRADIENT: 00:00:00.1290261
2:47:37 PM  THRESHOLDINV: 00:00:00.0002772
2:47:37 PM  ABSDIFF: 00:00:00.0029256
2:47:37 PM  Rp: 00:00:00.0421933
2:47:37 PM  CLOSE: 00:00:00.0014507
2:47:37 PM  Rv: 00:00:00.0335747
2:47:37 PM  SMOOTH: 00:00:00.0010389
2:47:37 PM  ------Evaluated. 00:00:00.2326165 seconds taken---------
The gradient function takes the morphological gradient of an image which is the difference between the erosion and dilation of an image. Erosion of an image will set pixels around the edge of objects in an image to 0. The erosion is done ten times in this filter. In image processing, an edge is an area in an image where the pixel intensities very greatly between adjacent pixels. For example, Figure X shows an image of a cat. Figure X shows the ten iteration erosion of the cat image. Note how the cat has been turned into a dark blob. All of the edges have been turned black.

Dilation is essentially the opposite of erosion. Instead of the pixels turning black, they instead turn white. Figure X shows the ten iteration dilation of the cat image.

The morphological gradient, as explained above is the difference between the two images. This is shown in figure X. It is easy to see how this process would take a long time. This one function actually calls twenty one functions, ten erosions, ten dilations and one subtract function. Therefore it is understandable that it would take 0.12 seconds.
After removing the GRADIENT function from the fast runner, the filter was run again. It took much less time to process one frame, halving the processing time. However, even though the filter took 0.1 seconds, the filter no longer worked. The GRADIENT function drastically changes the input image which creates very different output images for the rest of the nodes that use that image.

![Figure 29: Left, "Fast" filter output. Right, RepCalc input](image)

<table>
<thead>
<tr>
<th>Time</th>
<th>Description</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:49:42 PM</td>
<td>Rq:</td>
<td>00:00:00.0223646</td>
</tr>
<tr>
<td>2:49:42 PM</td>
<td>GRADIENT:</td>
<td>00:00:00.0006910</td>
</tr>
<tr>
<td>2:49:42 PM</td>
<td>THRESHOLDINV:</td>
<td>00:00:00.0002354</td>
</tr>
<tr>
<td>2:49:42 PM</td>
<td>ABSDIFF:</td>
<td>00:00:00.0004473</td>
</tr>
<tr>
<td>2:49:42 PM</td>
<td>Rp:</td>
<td>00:00:00.0423385</td>
</tr>
<tr>
<td>2:49:42 PM</td>
<td>CLOSE:</td>
<td>00:00:00.0014486</td>
</tr>
<tr>
<td>2:49:42 PM</td>
<td>Rv:</td>
<td>00:00:00.0342547</td>
</tr>
<tr>
<td>2:49:42 PM</td>
<td>SMOOTH:</td>
<td>00:00:00.0011196</td>
</tr>
</tbody>
</table>

2:49:42 PM  -----Evaluated. 00:00:00.1034130 seconds taken---------


10. Discussion

The results of the first experiment have shown that it is very hard to predict the functions that create the filters. The filters evolved from the same datasets were very different, only sharing one or two functions with other filters. In all of the cases, the best filter of the bunch was drastically different to the other filters. In two of the three cases, the best filter was evolved in the smallest amount of time. The large difference between each filter means that a much larger set of filters need to be used. The evolution process should be run for a specific amount of time so that each filter has an equal chance to evolve. The results given from this test give an impression that there are some functions that will lead to a better filter than others but the small data size hindered the ability to conclude what those functions were.

The second experiment was a success. The filters that were evolved on the Linux machine worked just as well as their Windows counterparts. Once the code worked there were no problems with it. The parameters did not need to be changed, image functions did not need to be altered very much and the only functionality that had to be removed was a simple visualisation of the change in fitness over time.

The third experiment opened up a new avenue to possibly improve the evolution process of CGP-IP. When looking at a filter created by a human, the processing time of that filter will be as low as possible because the person would have spent a lot of time optimizing their code. However CGP-IP does not do this. There is no kind of speed metric used in the evaluation of an individual’s fitness. This means that an individual that is slow to process has the same weight as an individual that is quick to process.

A possible improvement to the evaluation process of CGP-IP would be to introduce some kind of weighting to the processing speed of an individual. Possibly some kind of exponential function that decreases fitness a lot at high processing speeds and as the processing speed decreases, the decrease in fitness is small. A simple threshold for speed may also prove useful, rejecting any individual that is slower than a certain speed.
This addition to the fitness evaluating process will introduce issues into the system, namely an increase in the time taken to evolve a good individual. The effects of this will have to be explored alongside the possible new implementation.
11. Conclusion

Cartesian Genetic Programming has been proven to be a very successful way to approach machine learning. It has been built upon a previously very successful approach and has achieved a very useful trait, generality. This generality lets it be used for many different situations. It can be used to develop a movement algorithm for a robot, it can be used to effectively create image segmentation and noise filters and it can even be used as a part of a bigger process to help a robot learn almost by itself.

A lot of the time spent on this project was to get the code to run properly. Having never seen code as complex as CGP-IP before, semester one involved a lot of trial and error. The use of an input file to set parameters was not something that had been experienced before and several methods were passed over when trying to analyse the code. But with the help of the supervisor, who knew about and wrote the code, all difficulties were overcome and the code was able to run on a windows machine.

With the code working, semester two was about two main themes. (?) Attempting to learn about the process of developing a good (and not so good) filter and translating the code to work on a Linux OS. The use of Linux was only a recent endeavour and there was little knowledge about the differences between Linux and Windows. From downloading/installing programs to what worked and what didn’t, it was all different. As the translation from Windows to Linux progressed, a better understanding of the code was also developed. It became very easy to locate important parts of code and follow the flow of the program.

The translation was a success and satisfactory filters were evolved on a Linux machine. The filters were just as good as the ones developed on the windows OS. The stochastic process of evolving a filter made it hard to discover a lot of information about what makes a good filter. There was a fair amount of information on what functions were less useful though.
12. Bibliography


