NTNU
Norwegian University of Science and Technology

Creating Images with Self-Modifying Cartesian Genetic Programming

Christoffer Tønnessen

Master of Science in Computer Science
Submission date: July 2016
Supervisor: Gunnar Tufte, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Self-Modifying Cartesian Genetic Programming (SMCGP) is a form of genetic programming where the program can modify itself. This ability to modify itself is thought to make the program do more complex calculations from a simple starting point. Since its creating SMCGP has been used to solve a smaller set of problems [HMB11] and its ability to self modify has given good results.

In this thesis SMCGP's ability to create images is explored. This is done by having an evolutionary algorithm use SMCGP to look for images. Images are split into four groups: single colored image, simple pattern image, complex pattern image and random noise image, and the system is trying to create an image for each group. Should this be possible, then the system has shown it can create a wide variety of images with different properties. The self-modification part is explored by seeing if there's a correlation between the initial graph size and the final product as well as taking a look at whether this property helps evolution or not.

A framework to create images using SMCGP has been created for this thesis and all experiments that have been run have used this framework.

Experiments show, that by having a simple fitness function to tell the framework if the graph produces and image that is closer or further away form a desired result, images in all four groups are easily found. The ability for the graph to self-modify does seem to help with evolution. These experiments were run with graphs that had different amount of nodes in them at the start, and even the smallest graphs were able to create all desired images. This shows that the initial graph size is not the defining factor in the complexity of the resulting image.

Sammendrag

Selvmodifiserende Kartesisk-genetisk programmering (SKGP på norsk, SMCGP på engelsk) er en type genetisk programmering hvor programmet kan modifisere seg selv. Denne muligheten til å modifisere seg selv er tenkt å gjøre programmet bedre rustet for å klare komplekse kalkulasjoner når utgangspunktet ikke er komplekst. Siden dette sysemet ble laget har det blitt brukt til å løse noen problemer [HMB11] og systemets evne til å modifisere seg selv har gitt gode resultater.

I denne avhandlingen undersøkes SKGP-s evne til å generere bilder. Dette er gjort ved å bruke en evolusjonær algoritme for å finne bilder. Bildene har blitt splittet i fire hovedgrupper: ensfarget bilde, enkelt mønster på bilde, komplekst mønster på bilde og tilfeldig støy i bilde, og systemet prøver å finne minst ett bilde i hver gruppe. Skulle dette være mulig vil det ha vist at systemet kan lage bilder med flere forskjellige egenskaper. Selfmodifiseringsdelen blir utforsket ved å se om det er noen sammenheng mellom startstørrelsen på grafen som lager bilder og bildet som blir laget. Det blir også undersøkt om selfmodifiseringsegenskapen hjelper til med evolusjonen som finner disse bildene.

Et rammeverk for å lage bilder ved bruk av SKGP har blitt laget for denne avhandlingen, og alle eksperimenter som blir kjørt bruker dette rammeverket.

Eksperimentene viser at det å bruke en enkel fitnessfunksjon til å fortelle om et bilde er nærmere et ønsket bilde eller ikke, er nok til å generere bilder i alle fire grupper. Evnen til å modifisere seg selv ser ut til å hjelpe til med evolusjonen. Disse eksperimentene ble kjørt med grafer som startet med flere forskjellige antall noder, og selv de minste grafene klarte å lage alle ønskede bilder. Dette viser at startstørrelsen på grafene er ikke den avgjørende faktoren i kompleksiteten til bildene som blir generert.

Acknowledgements

I would like to thank my supervisor Gunnar Tufte for guiding me through this thesis.

I would also like to thank everyone close to me who have supported me immensely.

Contents

List of Figures

List of Tables

List of Acronyms

CGP	Cartesian Genetic Programming.
EA	Evolutionary Algorithm.
Fitness Function	A function used to measure performance of an individual in EA.
NTNU	Norwegian University of Science and Technology.
SMCGP	Self-Modifying Cartesian Genetic Programming.

Chapter 1 Introduction

Many parts of nature shows signs of self-organization. Repetitive and random patterns show up everywhere and they have all been created seemingly without guidance [H+01]. These small patterns form simple rules for simple creatures with no apparent rules for a bigger, more complex system. There seem to be no one guideline that watches over everything and has steered evolution in any direction, yet these complex structures exist. This is in great contrast to human engineering, where most, if not all, are defined by rules that take the bigger picture into consideration from the start. This is because human engineering in this regard uses a more of a top-down approach to solving a problem rather than a bottom-up approach [DSM13].

Biological organisms that make up bigger systems like swarms are examples of systems that are both self-organized and show tendencies that make them seem architectured [DSM13]. Each individual shows little knowledge of the bigger picture, and if separated from the group will not be able to show a great deal of intellect. However when each individual is a part of the bigger swarm the swarm as a whole shows complex emergent behavior by local interactions [DSM13]. This emergent behavior is interesting because it shows that non-architectured systems can create results that is similar to architectured systems. This shows that one can create a great and complex system from simple rules, which will be very scalable. A simple structure can produce a complex output as long as there is data transfer between the participants of the structure. This is because each individual can do a small amount of computation and pass on their result, when enough computation has been done, the result is combined to produce the complex result.

This can be related to how a computer and programming works. While a programmer often writes in a high-level language with concepts that makes sense for a human, a computer will deconstruct these commands into very simple, almost atomic, commands that will be executed.

Current ways of creating software by human design, is approaching the complexity ceiling, where we are approaching the upper limit on complexity of software [MT04].

While software that can be constructed using current methods of creating software can do some immense things, these are nowhere near the complexity of living systems [MT04]. To be able to break the complexity ceiling and make more complex software, one will need to mimic the biological development of multi-cellular organisms [MT04].

One big part of biological systems is that they can adopt to their environment [H^+01]. This is done by having a number of stable stats for a system that is large enough to react to all perturbations [H^+01]. *Self-Modifying Cartesian Genetic Programming* (SMCGP) is a type of genetic programming. This is an extension of Cartesian Genetic Programming (CGP) [Koz92] with the main different being that SMCGP allows for self-modification. What this self-modification allows for is to let evolution modify extra parameters of the system, which hopefully can provide a way for find a small and simple structure that over time can produce a large and complex structure. This could make the system better at reacting to perturbations as well as giving the evolution another parameter to change which can change the system immensely.

In this thesis a system for generating images using SMCGP has been created. SMCGP was first introduced in 2009 [HMB09] and has not been used for many things since then. There have been some experiments that show the basic capabilities of SMCGP, as well as compare them to regular CGP with very favorable results for SMCGP[HMB11]. One of these experiments touched on the possibility of creating images with SMCGP and this path will be further explored in this thesis. To achieve this task a framework for working with SMCGP has been created and used for generating images.

Three main questions will be answered in this thesis:

1. Is SMCGP able to generate both simple and complex images, if so what types of images?
2. Does SMCGP's self-modification help in evolution?
3. Does the initial graph size have an impact on the images able to be created?

To answer these questions several experiments were done. Four groups of images have been categorized: single colored images, repeating patterns in images, compound images with multiple properties and fully random images. Images in all of these categories were found, some easier than others. The discussion in Chapter 5 goes into greater detail on what these result mean.

1.1 Thesis Structure

This thesis is structured in the following way. Chapter 2 provides background information on evolutionary algorithms, genetic programming, Cartesian genetic programming and self-modifying Cartesian genetic programming. Chapter 3 gives an overview of the system used for the experiments and the methodology used for the experiments. Chapter 4 shows the results of each experiment that was run and the setup for those experiments. Chapter 5 discusses the results and what impact these results have. Chapter 6 concludes the thesis and mentions what the future work for this field can be.

Chapter 2 Background

2.1 Evolutionary Algorithms

Evolutionary Algorithms (EA) has been around for a long time and has taken various forms up through the years [MHT14]. The main idea behind it is to mimic evolution by having a pool of individuals with certain unique properties. Each individual or a pair of individuals will then create offspring which are close to their parent, but not entirely similar. The properties of these offspring are measured, and the best ones will be the next generation's pool.

The power of evolutionary algorithms is the ability to efficiently explore large solution spaces. In contrast to real life, evolutionary algorithms on the computer can go through tens of thousands of generations of evolution in a small amount of time. For each individual the algorithm is told if it's closer or further away from a desirable solution.

One implementation of an evolutionary algorithm is to start off with one individual. This individual's performance is measured by some means. The function that measures this performance is often called a *fitness function*. Then some number of offspring is created from this one individual. The offspring are an exact copy of the first individual, but there's a chance some part of their properties are mutated according to some mutation rate. The mutation can be anything, and it can cause problems which will cause the system to crash or similar. When all offspring are created, they are each measured with respect to the first individual and each other. After everyone is checked the best one available is chosen to have as many offspring as the first individual, and the process repeats itself. This is continued on until a satisfactory performance is found or a hard limit of some sort, often amount of evolution, is reached. One will then end up with the best solution found.

Since the technique used for finding solutions is based much off of randomness, it is an idea to run multiple parallel experiments at the same time. Doing so will increase the chances of finding a good individual that satisfies the demands set by the system, since more solutions are checked.

2.2 Genetic Programming

Genetic Programming (GP) [Koz92] is an evolutionary algorithm-based method [GH88] to create programs that fulfill a user-defined task. This is more often than not a series of computations on numbers which tries to come up with a result that is satisfactory. In GP the computations is called a program and is most often represented by a tree [Cra85] as shown in Figure 2.1. In such a tree each node will either have a function that manipulates values, or will have constant values. If the node is a function node, it will use its child nodes as input values, execute its function and pass on the result to its parent. Each node of the tree is executed one by one.

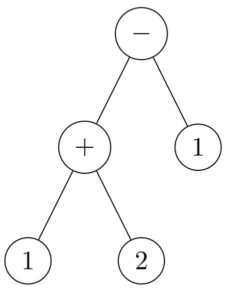


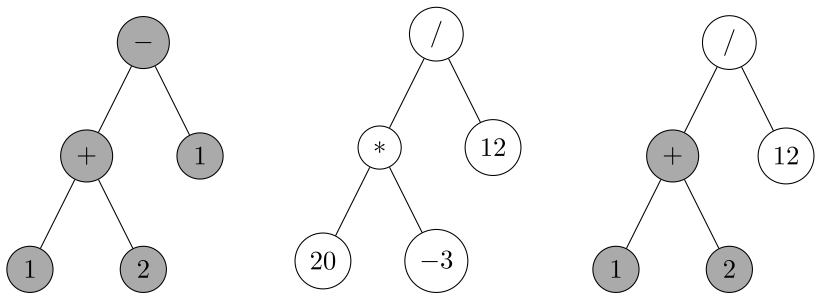
Figure 2.1: A program in Genetic Programming as a tree. Each node has either a value or a function. This program is equivalent with the mathematical calculation $(1 + 2) - 1$.

To find solutions in GP an EA is run. A population of random programs is generated and executed. The results of these programs are measured by the fitness function to see how well they are performing. The better the fitness, the closer the program is to completing its task.

The usual way to do evolution in CGP is called crossover evolution [Koz92]. This system works by starting off with a large population of random individuals. Each individual is checked with respect to a fitness function to see who's the best individuals of their generation. When this is done two and two individuals are paired together, either the best ones or one of the best of the best with the lowest scoring individuals of the best. Then one part of the graph from one individual is merged with one part of the other individual to create a new individual, see Figure 2.2 for a visual example.

2.2.1 Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) is a form of Genetic Programming. In Cartesian Genetic programming one has moved away from representing the program as a tree and instead represents it as a combination of rows and columns of numbered nodes [MT00]. Because of this, the implicit structure of the program is a graph



(a) Graph for individual one. (b) Graph for individual two. (c) Their child using crossover evolution.

Figure 2.2: An example of crossover evolution with two individuals creating one offspring.

instead of a tree. While early work with CGP used both columns and rows, the number of rows quickly ended up being just one, which effectively reduces the program to be a list of numbered nodes [HMB11]. This numbering of the nodes is what gives the name "Cartesian" to CGP; nodes are numbered in the same way you can number anything on a Cartesian plane. Since the list is ordered, the nodes in the list will have a specific order of execution, which will guarantee that a each program has one, and only one way to be executed.

Another way of representing a CGP program other than a list is as a directed asyclic graph, as there is a direct mapping from the list of nodes to a graph. One motivation for wanting to use a graph as representation for a program instead of a tree is that a graph is more general than a tree [MT00].

Each node in a CGP program consists of a function, denoting the purpose of the node, and what connections the node has, this means that the genotype of a node in CGP can be as simple as a list of numbers which maps to each of these things. This simple system with using only numbers makes it easy to do evolution programmatically, as one will only need to split two lists and merge one part from one list with one part of the other list.

0	1	2
FUNCTION	CONNECTION	CONNECTION

Figure 2.3: A possible representation of a node in CGP. This node has two connections. Each number is translated into either a function to be executed or a connection to another node.

Figure 2.3 shows one configuration of a genotype for a node in a CGP program. This node consists of one function and two numbers denoting what other nodes this node is connected to. The function part of the genotype denotes what type of node it is, this is often implemented by having a list of possible functions for the node and using the function number as a lookup in this list. The function node will execute the function of the node with the inputs it gets from its connections and pass on the result to any node that it's connected with it. If the node is at the start of the graph, it will get its values from the input to the graph. The connection part of the node says what other nodes this node is connected to. There can possibly be more than two connections to a node [MT00].

To start off the program a couple of inputs is needed. These are represented as one or more values at the far left of the graph that is fed into the first nodes of the graph. A full execution of the program will be going through each node one at a time and execute its function then give the results to its connected nodes. Every node has the same amount of inputs, and only one output. What node will give the output for the whole graph can be specified, but often it is the last node of the graph. If the output node of the graph is not the last node its value will still be passed on to the other nodes and they will do their calculations.

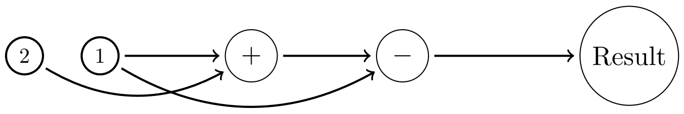
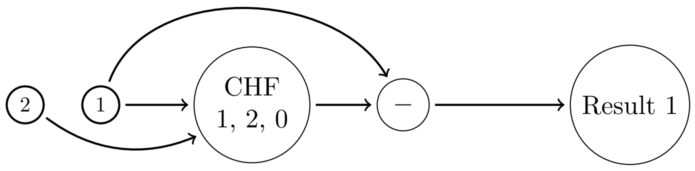


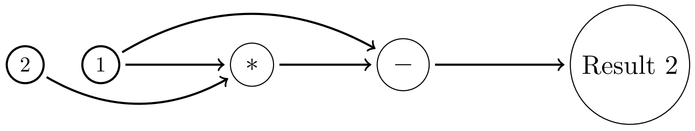
Figure 2.4: A CGP program where the first two nodes denote the input to the graph. Each node is executed from left to right. Each node takes as input the values passed to it, and passes its output on to the nodes they point at. The last node is the output of the program, denoted by "Result". This program will output the number 2 and can be represented by the mathematical function $(2 + 1) - 1$.

Figure 2.4 shows a CGP program which is equal to the tree program in Figure 2.1. One great benefit with CGP is that one can reuse old values that have been computed before. This would involve adding an extra edge where the extra value wants to be used. The node with the value 1 is used twice in CGP (Figure 2.4), but is duplicated in GP (Figure 2.1).

When evolving CGP there can be a lot of offspring that will produce the same result as the individual they are evolved from, but still have a different genotype. This happens for instance if a subgraph that produces an output which is fed into the first input of a commutative function is moved to be fed into the second input instead (like $2 + 1$ versus $1 + 2$). CGP can tackle this very well by having a concept of nodes that contribute to change of output, called active, and nodes that do not contribute to change in output, called inactive. The details of this is shown in [GP13], but the takeaway is that one can be very efficient with evolution by not recalculating values



(a) A SMCGP graph with a "change function node" with its parameters. This is the first iteration of the graph.



(b) After the first iteration is done, the "change function node" has changed its function to a multiply function, giving a different output than the last iteration.

Figure 2.5: Two iterations of a SMCGP graph to create two different results. The "change function node" (CHF) changes its function after the first run.

that are not needed to be recalculated. This will also guide the evolution process towards creating evolution that contribute to change in output.

2.3 Self-Modifying Cartesian Genetic Programming

Self-Modifying Cartesian Genetic Programming (SMCGP) is an extension to CGP that includes the possibility for self-modification in the execution graph [HMB11]. This is done by expanding the functions a node can have to also include functions that edit the graph itself. These types of functions is called functions are called overwrite functions while the functions that do not change the graph is simply called normal functions.

In SMCGP each node is specified by 6 genes [HMB11], one for the function of the node, two for which other nodes the node is connected to and three parameters required for some of the functions. The function gene works as with CGP where it maps to a function that denotes what type of node it is, but it's expanded to include self-modifying nodes. The connections in SMCGP are relative to the current node, and say how many nodes back the current node is connected to. If the connection is 1, it will connect to the previous node. A connection must be a number which is not 0, since the node can not be connected to itself. Should the connection point outside of the graph, then that is treated as being connected to an input in general SMCGP, this has been altered in the system created and is explained in Section 3.5. A node can be unconnected to another node in SMCGP, as the graph might change after being run a couple of times, which could connect the node later on. If a node have no other nodes connected to it, then it will not be executed and will need the self-modification to connect it later on for it to be executed. A node in SMCGP has always two inputs and one output, and the last node in a graph will be outputting the output for the whole graph. The parameters of a node is given to some of the functions if needed, and is mostly used in the self-modifying nodes and other nodes that aren't directly calculating a value. Figure 2.6 shows each part of a node in SMCGP.

0	1	2	3	4	5
FUNC	CONN	CONN	PARAM	PARAM	PARAM

Figure 2.6: A SMCGP graph node gene. The first value denotes the function, the next two is two numbers denoting which nodes the given node is connected to relatively to the node and lastly there are three parameters for the node.

All values in the node is checked to see if it is invalid, for instance if a node says it should connect with a node that's not a part of the graph. Should the value be invalid, then the system will take the modulus of the value until it satisfies a range of values that are valid. This makes for a resilient system where all values for a node is transformed to a valid value. The relative values are made to make it as easy as possible to duplicate and move subgraphs [HMB11], which makes self-modification easier. Figure 2.5 shows an example of a SMCGP graph.

Evaluating SMCGP graphs

Evaluating SMCGP graphs starts by feeding the input to the nodes that needs it, and then evaluating the nodes one by one, this is in a similar fashion to how CGP does it. The evaluation will happen recursively from the last node and backwards, fetching all the values that is needed [HMB09]. This way, only the nodes that are connected to the end through nodes to the right of it, will be executed.

For function nodes that process data, like $+$, $-$, $*$ and so on, the input values are passed to the function and the output value is the result of the mathematical function. For modifying nodes, first the input values will be checked, if the first value is less than the second value, there will be no modification of the graph and the execution of the graph continues. If the first input is greater than the second, the modifying function in the node will be added to a todo list of functions that should be executed. After the each node in the graph has been fully executed, the todo list is scanned and the modifications are done to the graph. The first modification that was added to the list is executed first, i.e. a first-in first-out system [HMB11]. To avoid the list growing too big in size, a limit can be introduced to limit the amount of changes allowed in one run. Some of these modification nodes will need the parameters that are encoded in the genotype, and these values are passed along to all functions. A list of the node functions and their needed parameters are shown in table A.1 and A.2 in Appendix B.

2.4 Self-Organization and Generation

There are other systems that show self-organization and other similar principles as SMCGP. L-systems [RS80] are a type of formal grammar that is used to generate structures. This system was created to describe how different types of yeast, fungi and algae grows. The system works by having a small set of rules that define how growth is done, and then a series of runs are done to see how the system grows. For instance a system could consist of two letters, A and B. A set of rules can be that every A can become a B or an AB and every B can become an A. So if you start off with A, then the next run will become AB, the next run after that will become ABA, after 7 runs the result will be ABAABABAABAABABAABABAABAABABAABAAB.

This way of creating a simple set of rules that creates something complex over time by self-modification is something SMCGP also is able to do. Instead of having a complex rule to create a very complex result, one can only have a simpler set of rules which will one step at a time come closer to the complex result.

Another important part of SMCGP is the ability for a node changes from one node type to another. This is called cellular differentiation [Sla09] in developmental biology. Usually this means that one node can change from a general node to a more specialized type. This happens multiple times when developing a multicellular organism and is one of the building blocks of evolution.

Chapter 3 Experiment system

To work with image generation in SMCGP, a framework for SMCGP manipulation has been written in Python [pyt]. The system has been developed and run on a MacBook Pro (Late 2013 model) [mac] with OS X El Capitan [elc] as its operating system.

The framework is fed a configuration with information about the images that are being generated. It will then create a new graph based on the configuration and execute this graph. Each execution of the graph will generate a pixel for the resulting image. The framework will continue to execute the graph for as many times as there are pixels in the image. After an image is created the framework will save the image so it's available to the user of the image. If the framework is being asked to evolve an image it will need a function to say if an image is better than another one. It will create a graph and create an image, it will then check the image to say if it's better or worse than another image and will after a defined amount of tries save the best image it found.

As mentioned, the framework supports executing a graph, specifying a problem that should be solved and creating a list of changes that should be done to a graph, as well as execute this list to change the graph. There can be an arbitrary amount of functions defined that the graph can use, which can either change the graph or not change it at all. This means that the framework can also be used for simple CGP programs if the functions that change the graph are left out. Each run of the graph takes in a specific configuration which tells the system how each run should be run and what should be run, which simplifies testing multiple configurations of the same system. A full overview of the available configurations is shown in Table 3.1.

3.1 Executing the Graph

To start off an execution the framework needs to take in a configuration for the run. All available configurations are listed in Table 3.1, and all values are either optional or have defaults implemented in the framework. This configuration includes useful

Configuration	**Explanation**
Genes	If present the system will use this set of predetermined genes.
Evolution	Specifies if the evolutionary algorithm should be run.
Offspring	How many offspring each generation should produce.
Mutation Rate	Probability for a mutation to occur during the evolution.
Epsilon	Fault tolerance in the solution.
Initial Graph Length	How big the initial graph is.
Maximal Changes to Graph	An upper limit of the allowed changes to the graph during a run.
Maximal Evaluations	An upper limit of the maximum allowed calculations done to a graph during a run.
Runs	The amount of runs done in an experiment.
Image Dimensions	The dimensions of the output image.
Parameters	The amount of parameters for a node.
Max Arity	How many inputs each node can take.
Memory Size	How big the memory for the graph should be
Verbose	If verbose logging is on or off.
Problem Function	What function should be used to compute the fitness.
Function List	The total amount of functions a node can take.
Color	If the output image should have color or not.
Randomize Input	If the initial input to the graph should be random.

Table 3.1: All the available parameters to create different systems configurations. Each value is either optional or has a default value in the framework, so any configuration is accepted.

parameters like graph length, how many elements there are in a node, how many nodes there should be in the graph, how many inputs to the graph and outputs from the graph there are, a list of all the functions to map function number to function, and the actual graph itself. This framework is as general as possible, so there's a lot of possibilities to change the amount of inputs, outputs, parameters and so on. This means that using this as an SMCGP framework is a matter of locking some of the configurations. When creating a new run, the configuration is read by the framework, and several classes are spawned.

These classes handles everything from making sure each run is running correctly, to the evolution part of the graph, to checking if an individual is better than another or not. In the program, the graph itself is represented by four main parts: a list of genes that represent the graph, a list of results from the various nodes, a list of memory for the graph to read and write to and a second graph that acts as the list of changes that will happen to a graph after each execution.

Each gene in the list of genes is exactly as each node in SMCGP and consist of six numbers that represent what function is present in the node as well as two connections and three parameters. Figure 2.6 gives a graphical overview of a node in SMCGP. The list of results is created to be able to easily share values between nodes and cache them during the same run. Each index in the list of results corresponds to the index of the node in the graph. The list of memory for the graph is an implementation of the memory system in SMCGP. The nodes can take on functions to interact with this memory. Lastly the second graph starts off by being a copy of the initial graph. When a node that modifies the graph is encountered, the second graph will be immediately changed, while the execution continues on the first graph. If a node that resets the changes list, the second graph is reverted to the same state as the graph being executed on. The last thing that happens after an execution is that the second, modified graph replaces the graph that was executed on this run.

To increase performance of the graph execution, the "active" and "inactive" system from [GP13] has been implemented. If a node is connected to a path that leads to the end of the graph it is considered active, and will be executed, if it's not connected to such a path it is considered inactive and will not be executed. This is determined at the start of each execution of the graph, as well as after every change to the graph that might result in a change of active nodes. All nodes that are considered active are saved and a list of these are kept for increased performance, as nodes can be skipped if they are inactive.

3.2 Specifying a Problem That Should Be Solved

To solve a problem, a specific class for this purpose is allocated. The main function for this class is to set a function that will take in a set of values, and in some way give a number indicating how close these values are to a desired result. For instance a function can be fed data generated from the individual as well as proven data that the individual should be able to create. In this case the fitness function would calculate a difference which says how far off the data generated from the individual is to the proven data. Another way of using the fitness function is to come up with a mathematical function that determines different properties of the result. Both of these cases have been implemented and used in this system.

Another way to test how good a solution is is to provide a *truth function* which will give the correct answer to a problem given an input. In this case the function would give the same input to the truth function as the individual got, and would calculate some difference between the two answers. This could be how far off the solution is with respect to a tolerance or similar.

3.3 Running the Evolution

To run the evolution a runner class is a part of the framework. This class handles the evolution and acts as en entry point as well as a manager for the framework. The runner handles all the flow of data between the framework.

Firstly the runner gets the configuration (see Table 3.1 for a full list of configuration possibilities) for the run as an input. It then prepares all the other classes based on the configuration, this includes delivering the problem function to the correct class, setting how many offspring each generation in the evolution will have and so on. Should some required parameters be missing from the configuration, default values are added.

When all the settings are set, the runner runs the program according to the specifications. There are two main paths that can be taken by the runner, either the graph is pre-determined in the configuration, or the framework should create its own graph. In the latter case all parts of the graph will be chosen randomly. Since the framework has taken extra precaution to make sure the system is as resilient as possible, explained in detail in Section 3.5, any value for any part of the genome for each node in the graph, however some reasonable ranges for values chosen are being used.

When the graph is set for the individual the runner checks if the evolutionary algorithm should be chosen, or not. If the evolutionary algorithm should not be run, the runner will feed any defined input to the individual and pass that on to the fitness function. The result will then be an image which is created from the individual given an input. However, should the evolutionary algorithm be run, the runner will then continue on creating multiple offspring from this one individual and run the process again for each of these individuals. A global list of all individuals during one generation of the evolution will keep track of all individuals and their fitness score. When one generation has finished evaluating, the best individual from this generation is kept and used as the base for the next generation. This system only keeps one individual from each generation and only one individual is carried over to the next generations.

The configuration also determines the *mutation rate* for the individual. This number represents the probability of one of the offspring of an individual to have a mutation in its gene. The way this is implemented is that one random number in the genome of the one node will be changed to a random value. Every value is a valid value, so the mutation can take on any value. The same function that creates a random graph is used for the mutation, so the values chosen are not so big that they could cause a performance issue. Both functions, connections and parameters can be mutated.

Each run of the framework is completely isolated from every other possible run. This means that multiple instances of the framework can be run in parallel on the same computer with no issues at all. A common issue with running experiments in parallel is the initial seed to the random number generator being the same, but this is one of the issues that will not be an issue here because of the isolation.

3.4 Self-Modification

To run the self-modification of SMCGP in the framework a second graph is present on each individual. Each run of the graph can introduce numerous changes to the graph, and each of these changes are applied to the second graph. The modification happens in a first in first out order, so the changes are immediately applied to the second graph. Should the function of a node be a reset of the changes list, then the second graph will be reset to the current graph. After one execution of the graph is fully completed, which means one value is outputted from the graph, the second graph will be set as the current graph. This ensures that the correct changes are applied after each execution of the graph.

To ensure good performance and avoid endless runs of ever-growing graphs, a limit is put on the number of changes that can be applied to a graph. If the limit is hit, no more changes will be applied to the graph, and the run will be terminated. Doing this extra graph will introduce unnecessary overhead if a node that resets the changes to the graph is hit. There is nothing in the framework that takes this into account, so this overhead is accepted.

All functions of the graph receives a reference to the individual that represents the graph. This is done so that the functions that alters the graph can do so on the individual itself for convenience. If the node doesn't alter the graph, it simply ignores the individual and manipulates the other values given to it. If it does alter the graph no values for calculations are returned, but rather a status signal saying the change was either successful or not, with the latter throwing an error.

3.5 Resilience Demands of Evolution

When evolving graphs in this system, one of the first things that become apparent is the demands for safeguarding every single aspect of the evolution. Since the graph can evolve itself, a whole new dimension of evolution is present. A graph can explode and become too big to be handled by the computer, or implode on itself and end up being nothing. All of these cases have to be considered to make sure the program doesn't crash in the middle of an experiment. A big part of the evolution is to explore as much of the possibility space as possible to see what is best which means there will be a lot of these issues.

To combat this there are multiple safeguard throughout the framework. The nodes are connected by a relative number to each other, if the relative number points outside of the graph, it will loop around and connect to the last node of the graph. This is different from the original SMCGP paper where nodes pointing outside of the graph would connect to the first or last node of the graph, depending on what side the relative number missed on. The reason for this change is results from experiments where they did seem to produce more favorable results when the connections were looping around. Another safeguard is a modulo calculation on every parameter fed to a function which checks to see if this parameter is within a range of valid values. Should it not be within these values the number will be moduloed and then fed to the function. This check means that any value can be sent to any function. There is also a timed limit to how long one run of the graph can take, if the time limit is reached, the run will be deemed invalid and the system will move on to the next calculation.

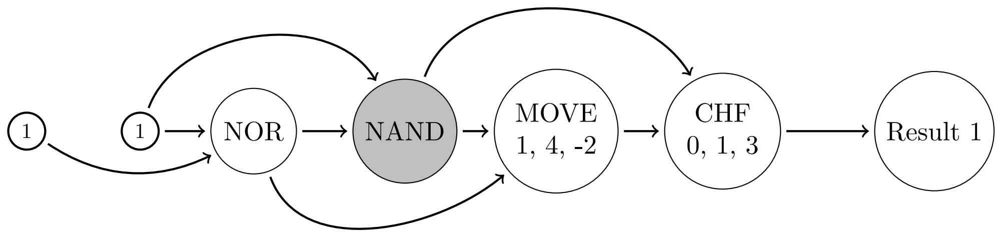
All of these safeguards makes not only life easier for the user of the framework, but also lets the evolution explore more freely without the program crashing and therefore stopping the evolution.

3.6 Self-Modifying Graphs as Image Generators

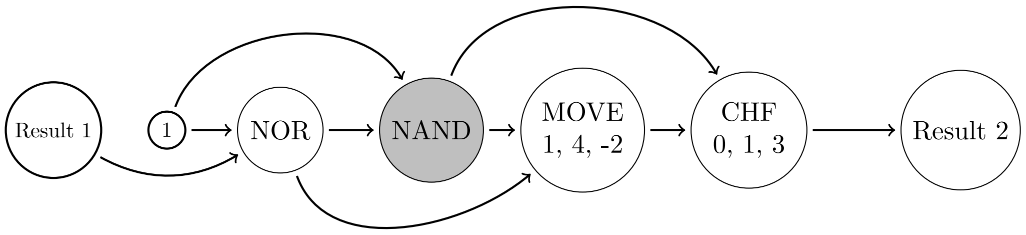
An SMCGP graph will produce one number from one run of the graph, and the modify itself. On the basis of this, the system that was chosen to create images were to have each run of the graph create one pixel for the image. This means that the graph will be run for as many times as there are pixels in the image. With the graph being able to modify itself in between each pixel generated, and the shared memory of the graph will not reset until the whole image is created, the system will hopefully be able to create images where one part of the image can be dependent on other parts of the image.

To start off the graph is given two inputs, which can be arbitrarily chosen, but the default values are 2, also chosen arbitrarily. When one iteration of the graph is complete, the result of the run will be fed back into the graph and replace the oldest of the two input values to the graph, see Figure 3.1 for a visual explanation. This is done to give the graph more possibilities to create interdependence between the different parts of the image.

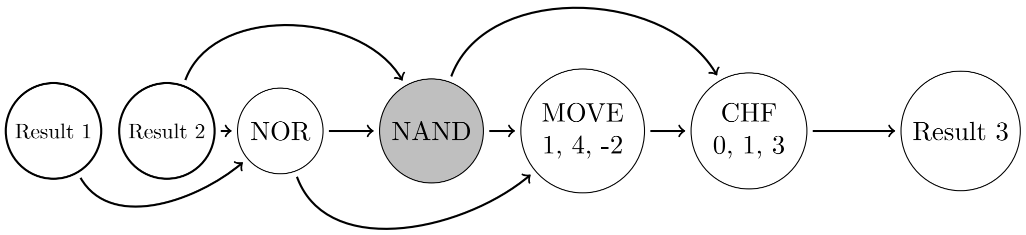
The graph itself have no concepts about the different properties of the image, like width and height. It will always create a sequence of pixels, and it's the rest of the



(a) An SMCGP graph will create a result from one run. The inputs to the graph is the first two nodes right now with the values 1 and 1.



(b) The result from the first run is replacing one of the input values for the next run.



(c) The oldest input value of the two inputs will always be replaced.

Figure 3.1: This is how the feedback system in the framework works. The graph starts out at its initial state and produces a result. This result is then fed back into the graph for its next run. The next result generated will then replace the oldest input to the graph.

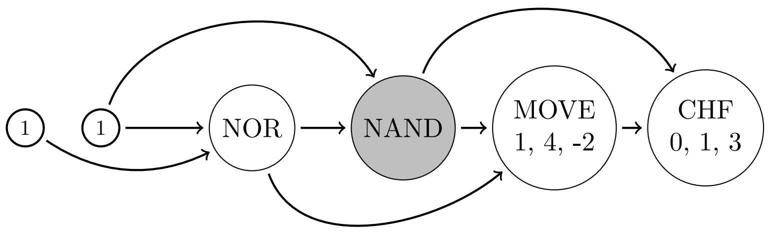
framework's task to arrange these pixels in the correct format to create an image. This leaves more work to be done by the fitness function to introduce these concepts to the final result. Therefore the fitness functions needs to have a concept of image height, width and other properties the final image should show The fitness function will be fed a long list of pixels generated from the graph and analyze this. For instance, a fitness function to create a French Flag [Wol68] would divide the list of pixels into rows with the length of the image width. Afterwards the contents of each row will be scanned and split into three equally sized groups. Then the contents of each group will be scanned to determine if it's the same color of each pixels or not. If there's a threshold present this will be taken into account when examining the contents of each group. Finally a check will be done to make sure none of the three groups contain the same color. This will be repeated for each row, and the wanted colors could either be pre-determined, or one could say that the first group will determine the colors for the flag. Afterwards a fitness needs to be returned, which can for instance be how many percent of the rows are the correct colors within the given threshold.

The graph itself is operating on numbers, but the output of the graph should be a pixel value, so a mapping from number value to a pixel value is present. There is a fear here that the mapping will introduce a bias in the system, and this is discussed more in-depth in Section 3.7. Each pixel generated is in the RGB format and consist of three numbers ranging from 0 to 255 for the red, green and blue channels. The way the mapping works is that the number is converted to binary form, and the first 8 bits are put into the blue channel, the next 8 bits into the green and the next 8 into the red. Since the graph operates in floating numbers the values must be converted into integers before the mapping to pixel values takes place. This is done naively where the decimals of the number is cut away. Another possibility is to scale the number up before doing this converting to capture more of the number before converting it into a pixel. Both possibilities were tested but there were no immediate advantage to any of the conversions, and the first one was chosen for this framework.

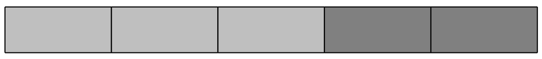
Figure 3.2 shows how an image is created with this system. A program is run to create a pixel and the values are fed back into the image, as shown in Figure 3.1. When a list of values are created they are ordered according to the specified properties of the image. When there's enough values for all the pixels, the image is complete.

3.7 Eliminating Bias in the System

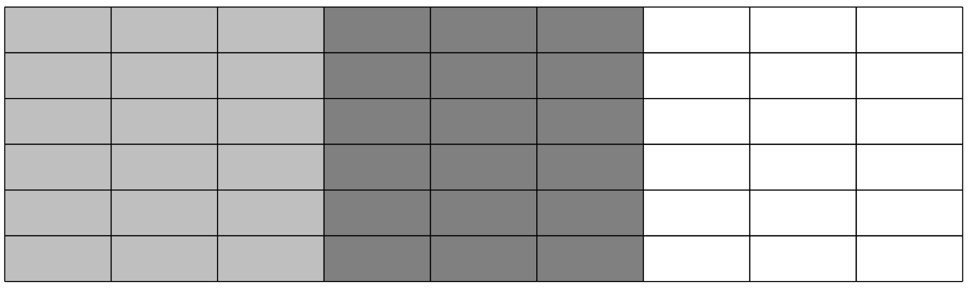
One problem of system like this is the possibility of a bias being a part of the system. If a bias should be present then it could mean that it's not the graph itself that is creating these images, but rather another part of the system which is unintentional. Since evolution has no concept of what is creating the images, it just gets values based on a graph, it can not know what part of the system is actually responsible for the way the image turned out. There must be a way to make sure that there's actually the graph that creates the images, and not the translation from floats to integers, or maybe the mapping from number to a pixel value that creates the spectacular images.



(a) An SMCGP graph will create a pixel for each run of the graph.



(b) These pixels are given in a sequence with no concept of how the end image is going to look.



(c) The framework makes sure to order the pixels according to the specifications of the image being generated. As shown here, the final result is a full image.

Figure 3.2: How the framework generates images. Firstly a graph is present that generates a pixel for each run. The graph will generate as many pixels as there are pixels in the resulting image. The framework itself takes this long list of pixels and puts them together with the correct width and height to create an image.

One of the famous papers on SMCGP [HMB11] created an SMCP graph to evolve the French Flag sequence. What the author did was to create integers that repeated a pattern, but treated some of the values created by the graph as "dead values". This meant that some of the values of the graph were thrown away without usage. Doing this could have introduced a bias in that author's system which could have altered the results in that particular experiment.

When creating this system to evolve SMCGP some biases were found. Figure 3.3 shows one of the biases found. In this picture the mapping from one big list of pixels to the actual image went wrong. Instead of splitting the list on the image width and place the rows underneath each other, the system put the rows both horizontally and vertically. This created a diagonal from the top left to the bottom right of the image where the image is mirrored over. While this property of mirroring is very interesting, it should be noted that the graphs and SMCGP did not create this effect, but it was the faulty mapping that caused this.

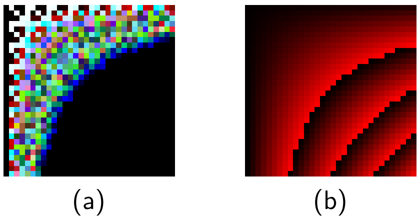


Figure 3.3: Bias shown in two images. When saving these images the framework read the same pixels multiple times and duplicated them on the same image. This caused the top side of the image and the left side of the image to be the same. This in turn made it look like the image had a twisting pattern.

Figure 3.4 had a similar problem with faulty mapping. In this case the system did not advance a full row in the list of pixels when saving the image. This caused each row of the image to be the last row plus one new pixel. The effects of this problem was a striped pattern throughout the image which was not caused by the SMCGP graph itself.

In both these two cases of bias the solution was to make sure the image was saved correctly when going from data structure in the framework to image on disk. A simple test to make sure this was done correctly was to read back the saved image after a save and check that the values for the pixels in the image, put together as one big list, was the same as the values for the image before being saved.

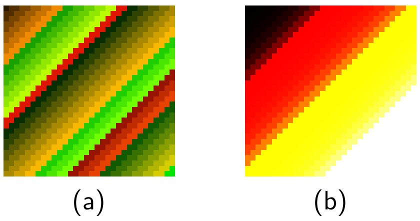


Figure 3.4: Bias shown in two images. When these images were saved the first row of the image saved onto all the other rows with a small offset. This caused the striped pattern, so the graph itself has not created this striped image.

Figure 3.5 shows a bias in the actual saved format. When saving the first images with this system these images were the result. The fault here is that the actual format being saved is wrong, which caused each pixel to have the wrong color. While this can look like interesting properties of the image, it is actually an image that doesn't correspond to the image being saved. To solve this problem a predetermined image was put into the system, and the save function was tested isolated from the rest of the system.

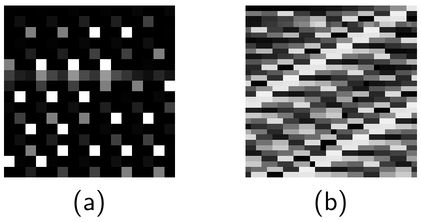


Figure 3.5: Bias shown in two images. These two images were corrupted when being saved. When the pixels were written to disk a bug occurred which caused some of the data to be written faulty. This in turn caused the patterns shown in these images. Had the saving gone correct, then the images would have been black.

To eliminate these biases a system were needed to prove that the graph itself were responsible for the image generation. No system were found that gave rigorous testing of a system similar to this one to make sure there were no biases. Having a good rigorous system is a challenging task, since there's a lot of moving parts in the system and a lot of places where a bias might be.

The system to test if there's biases in the framework consist of several tests. If an image passes one version of the test it is deemed highly likely that the graph is the main part in creating the properties of the image. This does not mean that there is a rigorous proof that no bias exist, but it does mean that there are some data pointing in that way.

When an image has been generated with an interesting property the graph that created said image is saved. Since this image was created with a given set of inputs as a starting point, this set of inputs is changed to something different. If there is a slight change in the output image, but it still retains its properties, for instance say the color changes, but the structure of the image is the same, it is deemed likely that no bias exist. Should the image not be able to do such a thing multiple runs of the experiment will be run. Should multiple runs turn up multiple images with the same structure, but with some slight changes, for instance that the structure is moved around, it is deemed less likely that a direct bias exist that created that one image.

These tests increases the likelihood of biases not being present in the system. The idea of the tests is to try to remove as much of the uniqueness of the one experiment being run, to see what general properties the graph has. All the results in Chapter 4 has passed one of these tests, with the exception of Section 4.4 which seems to be a special case of another experiment. In addition to these manual testing of the system has been conducted with to make sure for instance the pixels are being placed on the right places.

Chapter 4 Experiments

Four main types of experiments were performed to explore the ability of the SMCGP to create images. The background for choosing these experiments was to show that SMCGP is able to create a variety of different types of images.

The first experiment created a single colored image that changes color based on its inputs. The second experiment is an experiment to see if the system could produce repeating patterns. The third experiment was to create an image that is fully random and has no apparent pattern. Lastly an experiment was done to create an image that is in between total randomness and repeating patterns.

All of these experiments was done to test the capabilities of the system to produce different types of images. The configuration used as a base for all experiments is shown in Table 4.1. Any change that is specific to any experiment will be shown in its respective section.

Parameter	**Configuration**
Offsprings	100
Mutation rate	0.01
Epsilon	10
Initial graph length	4
Maximal changes to graph	1000
Maximal evaluations	1000
Image dimensions	32x32 pixels

Table 4.1: Default configuration of the system for the experiments done in this thesis. All values that are not present are left to their default values.

A total of around two million experiments were run in total, including runs that were done before biases were found in the system. The longest time an experiment run were for over 6 hours. The results presented here have filtered out all experiments that did not yield at least one image that was wanted. All tables displaying amount of correct results allow duplicate images, so if 100 images were created and two of them are identical as well as defined as correct solutions, it will be counted as 2 correct solutions.

4.1 Single Colored Image

Introduction

As a simple starting point to make sure that the graph is capable of creating images that rely on input an image that produces a single color based on its input. The configurations for this experiment is shown in Table 4.2.

Parameter	Configuration
Offsprings	100
Mutation rate	0.01
Epsilon	0
Initial graph length	1, 4, and 20
Maximal changes to graph	Unlimited
Maximal evaluations	Unlimited
Image dimensions	32x32 pixels, 64x64 pixels and 128x128 pixels

Table 4.2: The configuration used for the single colored image experiment. All values that are not present here are left to their default values.

Setup

The fitness function used for this experiment took the first pixel and declared this as the color of the image. The next pixels were checked against this one pixel and the fitness was calculated by how many percent of the rest of the pixels had the same color as the first one.

Several different sized images were created and several different graph lengths were tested.

Results

This task was easily solvable by the system. To solve this only one node is needed, for instance an NOP node which just passes its value on. This was also reflected in the results, as many of the graphs that produced a uniformly colored image based on the input was just one node.

Three runs were done to see if the graph could produce an image with a similar color. Each run had a different sized graph, with the number of nodes being 1, 4 and 20.

Every size produced an image with the same color. Since the whole image has the same color the output image can be any size as soon as one graph is found.

When the number of nodes in the graph increased, most of the graph was not being executed. The active part of the graph was no more than a node or two while the rest was either not connected to the rest, or didn't have an effect on the output.

The success rate for this result is shown in Table 4.5. To be counted as a correct solution in this experiment the whole image had to have one single color. This color was defined to be any color that was not black. The reason for this is that the initial values for the results of the graph is set to 0, so a graph that does nothing could be able to create a black image. This is usually not a big deal, since the graph will do a lot of work, but in this simple case the restriction seemed necessary.

Image Size	**Amount Created**	**Correct Solutions**	**Percentage**
32x32	80	44	55.00%
64x64	70	37	52.86%
128x128	100	59	59.00%

Table 4.3: The results for the single colored image experiment.

4.2 Repeating Pattern

To explore the abilities of the system two experiments to produce repeating patterns in the images was performed.

The first experiment was to create a predetermined pattern. This involved creating a fitness function that explicitly said how close one was to a given result. The results is shown in the subsection 4.2.1.

The next part of this experiment were to create another repeating pattern, however it should be more complex than the first pattern. In this case a more complex repeating pattern means the sequence can not simply be two fluctuating values. As the results in the subsection 4.2.2 will show this ended up being a gradientual pattern.

Both parts of this experiment had the same configuration shown in Table 4.4, but with two different fitness functions. This experiment was only done with two different image dimensions. These images were created in two different dimensions, 32x32 pixels and 64x64 pixels. When trying to generate bigger images the time it took to generate each image increased drastically being up to hours for each image. Because of this the sizes settled upon were 32x32 pixels and 64x64 pixels.

Parameter	**Configuration**
Offsprings	100
Mutation rate	0.01
Initial graph length	4
Maximal changes to graph	10 000
Maximal evaluations	10 000
Image dimensions	32x32 pixels, 64x64 pixels

Table 4.4: The configuration used for the for the repeating pattern experiment. All values that are not present here are left to their defaults.

4.2.1 Striped Pattern

Introduction

Firstly a simple set of repeating patterns was created with the color of the image being specified by the input of the graph. This was inspired by the "French Flag Model" [Wol68] which involves creating a striped pattern in an output image. In this case a striped pattern of two colors was found which changed color based on its input.

Setup

The fitness function for this problem did a check on how much of the image contained stripes and how much did not. It started off by checking the first four pixels to see if two and two of these was the same color. After this the functions keeps matching the rest of the pixels to the first four and gives a score based on how much of the image is stripes. A pseudocode implementation of this function is shown in Algorithm B.1 in Appendix B.

Results

Image Size	**Amount Created**	**Correct Solutions**	**Percentage**
32x32	200	16	8.00%
64x64	200	28	14.00%

Table 4.5: The result for the simple repeating pattern in an image experiment.

Some sample images created by this experiment is shown in Figure 4.1. Since this image has a pattern that repeats after 4 pixels, each image created with this graph, as long as its width is a multiple of 4, will look the same.

The success rate for this experiment is shown in Table 4.5. To be counted as a correct solution the image had to have stripes of two different colors throughout the whole image, as show in Figure 4.1.

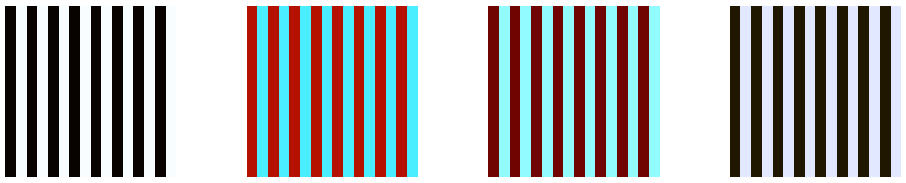

(a) Black and White. (b) Red and Light Blue. (c) Dark Red and Light Blue. (d) Black and Light Purple.

Figure 4.1: A collection of different colored striped images of sizes 32x32 pixels created in this experiment. All images are scaled to twice their size for increased visibility.

4.2.2 Gradientual Repeating Pattern

Introduction

Another task in regards to repeating patterns in to create a more complex pattern than the previous repeating pattern. What the resulting pattern should be was not defined beforehand, but it quickly became apparent after a few experiments that a gradient pattern was possible.

When the gradient was found, a fitness function was created especially to find gradientual patterns.

Setup

The fitness function used for this checked one and one row of the image at a time. It divided the given row into sections of three and took the average value of the pixels in each region. It then checked to see if the average color of each section is darker than their next section, with a threshold of epsilon from the configuration. For each section of the row that were darker than its next section a good score was given that was later normalized over the amount of sections the row was split into. After this all these values were added together and normalized with respect to how many rows there were. A pseudocode implementation of this function is shown in Algorithm B.2 in Appendix B.

Results

The results are shown in Figure 4.2. Since the pattern will repeat itself at a given point, any number of sizes of images is able to be produced from either of the graphs

Image Size	Amount Created	Correct Solutions	Percentage
32x32	200	8	4.00%
64x64	200	18	9.00%

Table 4.6: The result for the gradientual repeating pattern in an image experiment.

that created the result images. However since the repeating pattern has a period, not all sizes will look the same. If you double the image size, the height of the gradients will be cut in half. Therefore a graph can create an image of any size, but they won't look like upscaled versions of the same image, but rather as different images with slightly different properties.

The success rate for this experiment is shown in Table 4.6. To be counted as a correct solution the image had to have a repeating pattern that was more complex than two different colors. All the results shown here are various variations of a gradient pattern shown in the results in Figure 4.2.

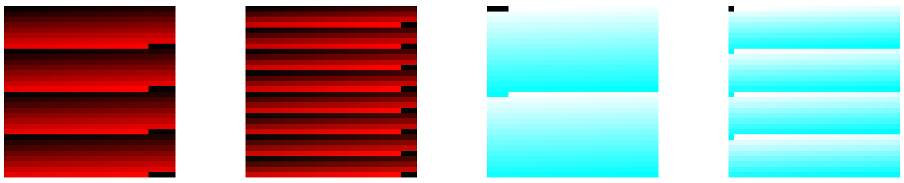

(a) Big, red gradientual stripes. (b) Small, red gradientual stripes. (c) Big, light blue gradientual stripes. (d) Medium sized light blue gradientual stripes.

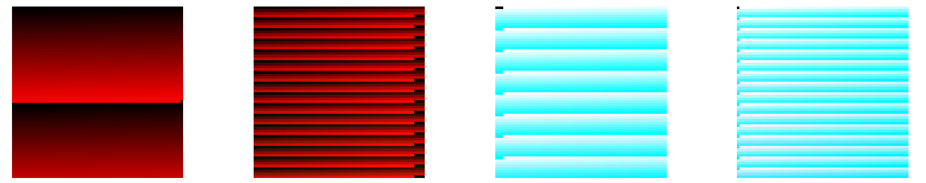
Figure 4.2: A collection of different images with repeating patterns of sizes 32x32 pixels created in this experiment. All images are scaled to twice their size for increased visibility.

4.3 Edge of Chaos

Introduction

To make sure the system is capable of creating more than a fully random image and simple repeating patterns, another image with different properties was wanted. The image should display a pattern that lies in between fully random and very repetitive. If this system can create an image with such properties it can be argued that the system is able to reach a phase where emergent computing is present [Lan90].

As the results will show, this other property became a noise-like effect. Like the gradient pattern task, this task was not predetermined. When running the other



(a) Big, red gradientual stripes. (b) Small, red gradientual stripes. (c) Medium sized light blue gradientual stripes. (d) Very small, light blue gradientual stripes.

Figure 4.3: A collection of different images with repeating patterns of sizes 64x64 pixels created in this experiment. No scaling is done.

experiments an image that had pillars with a noise-like pattern on it. Afterwards a custom fitness function was written and more experiments were done.

Figure 4.4 shows an image produced in which has pillars going downwards in the image, with a noise-like effect going though the whole image. As with the repeating pattern, when the images became too big, the runtime became too big to work with, because of this images with the size of 128x128 and up were not created. Images in sizes of 32x32 pixels as well as 64x64 pixels were created. Table 4.7 shows the configuration for this experiment.

Parameter	Configuration
Offsprings	100
Mutation rate	0.01
Initial graph length	4
Maximal changes to graph	10 000
Maximal evaluations	10 000
Image dimensions	32x32 pixels, 64x64 pixels

Table 4.7: The configuration used for the edge of chaos experiment. All values that are not present here are left to their defaults.

Setup

The fitness function used for this experiment was largely based on the fitness function for the repeating patterns, described in Section 4.2. The main difference between the two is that this fitness function did another pass over the image after the main fitness was found to see if the image contained small irregularities.

This pass took one and one row and counted each unique color in that row and gave a score that was higher if there were more colors in the image and lower if there were

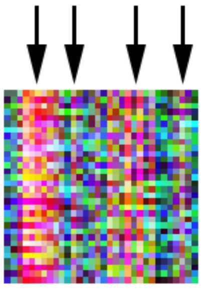


Figure 4.4: Image computed that shows signs of being on the edge of chaos [Lan90]. The arrows point at pillars running through the image, while the image as a whole shows signs of a noise-like pattern.

fewer. The sum was in the end normalized with respect to how many rows there were and how many pixels there were. At the end both values were given a weight to see how much of the total they should count. In this experiment the first pass with the repeating patterns were weighted at 30% and the last pass of random noise were weighted at 70%. A pseudocode implementation of this fitness function is shown in Algorithm B.3 in Appendix B.

Results

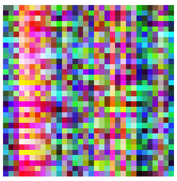
The results are shown in Figure 4.5 for the images of size 32x32 and Figure 4.6 for the images of size 64x64.

With these images there are some variation in the results from the smaller to the larger images. The smaller images have more clearly defined pillars which also takes up more of the image. The bigger images have more discrete pillars that aren't as clear to see. This is in contrast to the other experiments where there were close to no change in the different sized images.

The statistics for this result can be seen in Table 4.8. To be counted as a "Correct Solution" in this case means that the image had to visibly have pillars in the image, as well as looking mostly like a noise pattern. While there are some overlap, the results yielded different images with the wanted properties.

Image Size	Amount Created	Correct Solutions	Percentage
32x32	200	7	3.50%
64x64	200	3	1.50%

Table 4.8: Amount of correct solutions for an image that have pillars and a noise-like pattern.


(a) Big red pillar on the left, smaller pillars on the middle-right.

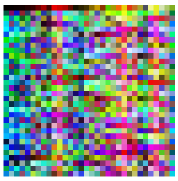
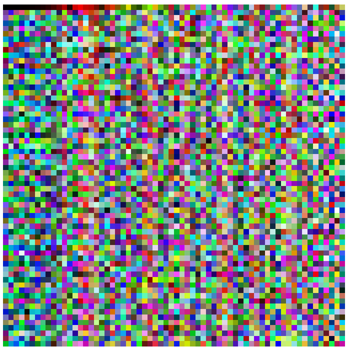
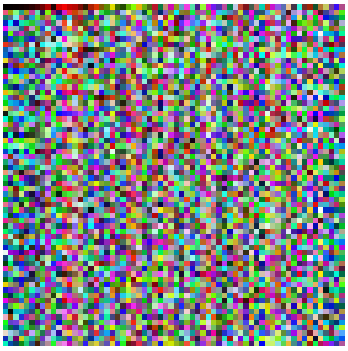

(b) Smaller red pillars on the right side of the image.

Figure 4.5: A collection of different images with a pattern seemingly on the edge of chaos [Lan90] of sizes 32x32 pixels created in this experiment. These images are scaled to twice their size for increased visibility.


(a) Smaller red pillars located on the right side of the image.


(b) Smaller red pillars located in the middle of the image.

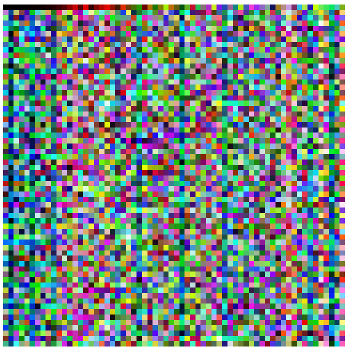

(c) Thinker red pillars in the center with two smaller on the right side of the image.

Figure 4.6: A collection of different images with a pattern seemingly on the edge of chaos [Lan90] of sizes 64x64 pixels created in this experiment. These images are scaled to twice their size for increased visibility.

4.4 Random Noise

Introduction

Another image that shows the capabilities of the system as a whole is being able to create an image with no apparent image at all. This ability can be seen as a way of creating a seemingly random sequence with no pattern at all. The results here were found when running the edge of chaos experiment from Section 4.3.

Setup

The setup for this experiment were exactly the same as the edge of chaos experiment. Each one of the 400 images created in that experiment were checked manually to see what interesting images were present. A few times an image popped up that looked like the it was fully random and this is the image that is presented here.

Results

The system were able to create an image that was seemingly random, however it was only able to create this image with an image size of 32x32. This might suggest that this result is a special case of a pillar image from Section 4.3. Only one form for this image was found, each of the results under "Correct Solutions" in Table 4.9 refers to the same image.

The results of this experiment can be seen in Figure 4.7. The statistics for this result can be seen in Table 4.9.

Image Size	**Amount Created**	**Correct Solutions**	**Percentage**
32x32	200	2	1.00%
64x64	200	0	0.00%

Table 4.9: The random noise image was found only when the image was 32x32 pixels, and there were only one variation of the image.

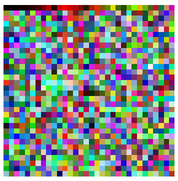


Figure 4.7: Random noise in image of size 32x32 created in the other experiments in this thesis. It is scaled to twice its size for increased visibility.

Chapter 5 Discussion

The results of the experiments show a variety in the properties of the images created. The same system, with different graphs are able to create several different patterns. This includes a simple single colored image (see Section 4.1), two different types of repeating patterns (see Section 4.2), complex patterns that shows multiple properties in one image (see Section 4.3) and even seemingly fully random noise in images (see Section 4.4).

5.1 The Framework

All the images were able to be created with a small initial graph length. With a graph of length 4, all types of images were found. This suggest that the graph length is not a defining factor when it comes to being able to produce images. Seeing it as a node can spawn multiple more nodes and other nodes can change their function this seems plausible. In nature a small cell can through cell differentiation and growth create a complex creature, and these results suggest this might also be present here. There might be one "base graph" that can turn into all other graphs, similar to how stem cells work in the human body. However, no such graph has been found in these experiments.

Each experiment, except for the experiment to find one single color for the whole image, was only done for images with the size of 32x32 and 64x64 pixels. The reason for this was that the runtime for each experiment became too long for any image bigger than that. The exact reason for this time increase is not known. One suggestion is that it could be that the search space for the evolution becomes so big when the image is 128x128 pixels, that a much more powerful computer is actually needed to solve these problems. Another possible reason for this is that the framework itself is not good enough optimized and may even have bugs which increases the runtime significantly. Both possibilities are plausible and it might even be a combination of the two that is the exact reason. While this did mean that bigger images were off the table, smaller images were still created, so principles were able to be shown.

As seen by the results, the images that were 32x32 have the same overall properties. The edge of chaos experiment have pillars and random noise in both the 32x32 and 64x64 images. The same goes for the repeating patterns as well as the single colored image. The odd one out is the random noise image, and this will be mentioned more later. Having all the same properties just scaled up with a bigger image suggest that the size of the image does not necessarily mean that the size of the image directly affects what can be generated in the image. One can say that images bigger than 64x64 pixels could have the same properties as the images seen here. This can not be said certainly, however, as the dataset for this idea is just two sizes. One would have to test a lot more images to see an exact trend.

While there has been a lot of work gone into making the possibility of a bias being in the system as small as possible, there is no way to be certain of this. A non-obvious bug in the system could pollute the framework in ways that are hard to tell. All the results here are subject to these errors. This can be implementation errors, framework errors, operating system errors and so on.

5.2 The Results

The system was easily able to create a single colored image from Section 4.1 of various sizes. A fully black image was deemed not a correct solution since a graph that did nothing will create a fully black image. The results shown in Table 4.5 shows how many of the images created was deemed a correct solution. This table shows that more than half of all images created were correct solutions. This high number makes it likely that the system and the evolution has worked together to find these images.

The repeating patterns in Section 4.2 have a much lower success rate. This means that a lot more images had to be created to find a suitable solution. It can with a reasonable be assumed that creating an image with a repeating pattern is harder than creating an image with just one color, so it makes sense that the success rate is lower here. Exactly how much less this number should be is hard to say. There are a lot of moving parts in this system, so pinpointing exactly what part impacts the results is not an easy task and have not been the focus of this thesis.

The success rate was higher for bigger images when evolving repeating patterns. This could be just random variation, but it could also mean that it is easier to find repeating patterns in bigger images. One reason that could support this theory is that bigger images have more in them, which makes it easier to find something repeating in them. The predetermined pattern were a shorter pattern than the gradient pattern, and this was found at a higher rate than the other. Based off of this one can assume that a smaller repeating pattern is easier to find than a bigger one, however the basis for this is only two experiments of two sizes each.

The edge of chaos images from Section 4.3 images were very rare to find. They showed up in only 3.50% of the images generated to find those images when the size was 32x32, and only about half of that when the size was 64x64. This can either mean that finding these images is very hard, or it could simply be pure luck that these images were found at all. Given that the percentage is low, but not insignificant, we can assume, but not say for sure, that these images are hard to find. This also fits with the other results that an image that has more in it shows up less when evolving images.

The random noise image from Section 4.4 only showed up two times when the experiment was to find an image on the edge of chaos. Given its resemblance to the edge of chaos images, and its rarity it might just be a special case of an edge of chaos image. There were no variations of this image found, just that one version, while all other images have been found in multiple versions. This suggest even more that this image is a special case of another experiment.

Chapter 6 Conclusion

The SMCGP system has shown the ability to create a vast array of different types of images. It was done so with a very limited starting point through various techniques inspired by biology.Both complex and simple images were able to be created by the experiments.

The evolution did seem to help along with finding these images with varying success rate. It is hard to estimate what the success rate should be for finding such an image, since there are a lot of moving parts in the system as well as calculating the probability of randomly creating this image is also complex. While there might be sheer luck that some of the images were found, and that the evolution part of the system might not have any part to do with this, it will not change the fact that the images were still found and therefore confirms that this system is capable of creating these images.

The single colored image was found almost immediately after the framework was completed. When the simple fitness function was complete the results were instantly either black images, or images of a different color.

The same goes for the striped pattern. While the actual amount of correct results was smaller the results were found after a few experiments and tuning of the fitness function.

When trying to find the edge of chaos images, it was a bit harder. Not only did the system have to run quite a few experiments, but the tuning of the fitness function was very important. The weighing of the different parts of the fitness function had a big impact on the results. After a series of testing the weighing shown in the results were found to produce the best results. This also led to the random noise image to be found.

All types of images were able to be found with a graph length of just 4 nodes. This is not the same size as the ending graph for all of the experiments, but it does say that the complexity of the output image is not directly tied to the graph length. This harmonizes well with the fact that the graph is able to increase its own size if needed and is not bound by its starting size.

One point to take into account is the fact that eliminating bias in this system is not an easy task. There have been some work done on eliminating bias and try to locate possible biases, but there is no rigorous test yet available. However the precautions done suggest the probability of there being bias in the system is less than there not being bias in the system.

The system does suffer from a significant slowdown when the images become big. This can either be because of bugs in the framework or because the search space becomes so big that more powerful computers are needed. However the two sized images that were created did show interesting properties in between them, notably that one could see the same properties scaled up when looking at the bigger images in contrast to the smaller ones.

The two images of different sizes show the same properties, but scaled according to how big the images are. While this does not prove that all images will have the same properties regardless of size, it does show that there is a trend starting.

Overall this system is able to generate a wide range of different images that show different properties related to these types of evolving systems. Its self-modification properties seem to assist the evolution when trying to find correct images, which is has also been seen in other experiments [HMB11]. The initial graph size is not directly linked up to the complexity of the resulting images.

6.1 Future Work

The problems experienced when the images become too big are interesting issues to tackle. It could either be an optimization issue, since there are a lot of data being moved around during execution, or it could be a more significant bug. While there is done work to cache values being calculated, there is done less work in regards to smart solutions when evolving the graph itself. Work in this area could provide valuable insights into how to efficiently change the structure of large data structures.

One very interesting part is to analyze what graphs create what images and see if there is some correlation between properties of the graphs and properties of the image. This could be for instance one subgraph that causes a certain color to appear, or another subgraph that creates a triangle at a given spot. If such a graph can be found this would open the possibilities to create a grammar that explains a mapping from graph to result.

To further explore the possibilities of SMCGP an interesting question is to see if these results are replicable with a graph that never grows more than a given amount of nodes. Right now there are no restriction on how big the graphs can be, only restrictions on how much calculation is allowed.

Lastly an interesting problem is to see if the mapping from graph result to pixel color can be done differently, and see if this has any impact on the system. Right now there has only been one mapping, but a different mapping could provide other properties.

References

[Cra85] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the First International Conference on Genetic Algorithms*, pages 183–187, 1985.

[DSM13] René Doursat, Hiroki Sayama, and Olivier Michel. A review of morphogenetic engineering. *Natural Computing*, 12(4):517–535, 2013.

[elc] Os x. http://www.apple.com/no/osx/. Accessed: 2016-07-04.

[GH88] David E Goldberg and John H Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2):95–99, 1988.

[GP13] Brian W. Goldman and William F. Punch. Reducing wasted evaluations in cartesian genetic programming. In Krzysztof Krawiec, Alberto Moraglio, Ting Hu, A. Sima Etaner-Uyar, and Bin Hu, editors, *EuroGP*, volume 7831 of *Lecture Notes in Computer Science*, pages 61–72. Springer, 2013.

[H^+01] Francis Heylighen et al. The science of self-organization and adaptivity. *The encyclopedia of life support systems*, 5(3):253–280, 2001.

[HMB09] Simon Harding, Julian Francis Miller, and Wolfgang Banzhaf. Self modifying cartesian genetic programming: Parity. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 285–292. IEEE, 2009.

[HMB11] Simon L Harding, Julian F Miller, and Wolfgang Banzhaf. Self-modifying cartesian genetic programming. In *Cartesian Genetic Programming*, pages 101–124. Springer, 2011.

[Koz92] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[Lan90] Chris G. Langton. Computation at the edge of chaos: Phase transitions and emergent computation. *Physica D: Nonlinear Phenomena*, 42(1):12 – 37, 1990.

[mac] Macbook pro. http://www.apple.com/no/macbook-pro/. Accessed: 2016-07-04.

[MHT14] Julian F Miller, Simon L Harding, and Gunnar Tufte. Evolution-in-materio: evolving computation in materials. *Evolutionary Intelligence*, 7(1):49–67, 2014.

[MT00] Julian F Miller and Peter Thomson. Cartesian genetic programming. In *Genetic Programming*, pages 121–132. Springer, 2000.

[MT04] Julian F Miller and Peter Thomson. Beyond the complexity ceiling: Evolution, emergence and regeneration. In *Proc. GECCO 2004 Workshop on Regeneration and Learning in Developmental Systems*, 2004.

[pyt] Python, about. https://www.python.org/about/. Accessed: 2016-07-04.

[RS80] Grzegorz Rozenberg and Arto Salomaa. *Mathematical Theory of L systems.* Academic Press, Inc., 1980.

[Sla09] Jonathan MW Slack. *Essential developmental biology.* John Wiley & Sons, 2009.

[Wol68] Lewis Wolpert. The french flag problem: a contribution to the discussion on pattern development and regulation. *Towards a theoretical biology*, 1:125–133, 1968.

Appendix A

Functions available in SMCGP

Function	Parameters	Description
NOP	None	Passes the first connection value to the output.
$+$	None	Returns the sum of the input values.
$-$	None	Returns the subtraction of the second input value from the first.
$*$	None	Returns the product of the input values.
DIV	None	Returns the first input values, divided by the second.
AND	None	Performs a logical AND of the input values.
OR	None	Performs a logical OR of the input values.
NAND	None	Performs a logical NAND of the input values.
NOR	None	Performs a logical ANOR of the input values.
CONST	Value	Returns the first parameter.
INP	InputIndex	Returns the (InputIndex modulo the number of inputs) input value.
READ	Address	Returns the value stored in memory location (Address modulo memory size).
WRT	Address	Stores the first input value in memory location (Address modulo memory size).
PRC	Start, End	Executes the subgraph specified by Start and End as a separate graph with the calling nodes input values used as the graph inputs.

Table A.1: SMCGP function set, taken from Table 8 in paper [HMB11].

Function	**Parameters**	**Description**
MOVE	Start, End, Insert	Moves each of the nodes between Start and End into the position specified by Insert.
DUPE	Start, End, Insert	Inserts copies of the nodes between Start and End into the position specified by Insert.
DELETE	Start, End	Deletes the nodes between Start and End indexes.
ADD	Insert, Count	Adds Count number of NOP nodes at position Insert.
CHF	Node, New Function	Changes the function of a specified node to the specified function.
CHC	Node, Connection1, Connection2	Changes the connections in the specified node.
CHP	Node, ParameterIndex, New Value	Changes the specified parameter and a given node.
FLR	None	Clears any entries in the pending modifications list.
OVR	Start, End, Insert	Moves each of the nodes between Start and End into the position specified by Insert, overwriting existing nodes.
DU2	Start, End, Insert	Similar to DUPE, but connections are considered to absolute, rather than relative.

Table A.2: SMCGP Overwrite function set, taken from Table 9 in paper [HMB11].

Appendix B
Pseudocode

Algorithm B.1 Fitness function for evolving stripes.

```
# assume ‘image‘ is a list of pixels as numbers, i.e. 0x0000ff for red.

function fitness_for_stripes(image):
  p1, p2, p3, p4 = get_first_4_pixels(image)

  # give 0.5 score for each pair that matches
  starting_score = (0.5 if p1 == p2) + (0.5 if p3 == p4)

  # check the first four pixels against the rest of the image
  rest_of_image_score = 0
  runs = 0
  while image.has_more_pixels():
    # get the next 4 pixels
    n1, n2, n3, n4 = get_next_4_pixels(image)

    rest_of_image_score += 0.25 if n1 == p1
    rest_of_image_score += 0.25 if n2 == p2
    rest_of_image_score += 0.25 if n3 == p3
    rest_of_image_score += 0.25 if n4 == p4

    runs += 1

  # normalize checks from the rest of the image
  normalized_rest_of_image_score = rest_of_image_score / runs

  # add the sums together and normalize
  total_score = (starting_sore + normalized_rest_of_image_score) / 2

  return total_score
```

Algorithm B.2 Fitness function for evolving gradientual repeating patterns.

```
# assume ‘image‘ is a list of pixels as numbers, i.e. 0x0000ff for red.

function fitness_for_stripes(image, epsilon):

  score = 0

  for row_index in image.get_amount_of_rows():
    row = image.get_row(row_index)

    first, second, third = row.split_in_three()

    x = average_value(first)
    y = average_value(second)
    z = average_value(thrid)

    score += 0.25 if x < y
    score += 0.25 if abs(x-y) < epsilon

    score += 0.25 if y < z
    score += 0.25 if abs(y-z) < epsilon

  normalized_result = score / image.get_amount_of_rows()

  return normalized_result
```

Algorithm B.3 Fitness function for evolving an image on the edge of chaos.

```
# assume ‘image‘ is a list of pixels as numbers, i.e. 0x0000ff for red.

function fitness_for_stripes(image, epsilon):
  score = 0

  # Part I: the same as the striped image one
  p1, p2, p3, p4 = get_first_4_pixels(image)

  # give 0.5 score for each pair that matches
  starting_score = (0.5 if p1 == p2) + (0.5 if p3 == p4)

  # check the first four pixels against the rest of the image
  rest_of_image_score = 0
  runs = 0
  while image.has_more_pixels():
    # get the next 4 pixels
    n1, n2, n3, n4 = get_next_4_pixels(image)

    rest_of_image_score += 0.25 if n1 == p1
    rest_of_image_score += 0.25 if n2 == p2
    rest_of_image_score += 0.25 if n3 == p3
    rest_of_image_score += 0.25 if n4 == p4

    runs += 1

  # normalize checks from the rest of the image
  normalized_rest_of_image_score = rest_of_image_score / runs

  # add the sums together and normalize
  total_stripe_score = (starting_sore + normalized_rest_of_image_score) / 2


  # Part II: random noise, new pass thorugh the image
  pixels_seen = {} # Empty dictionary: This dictionary will map from
                   # pixel color to amount of times that color is seen

  for pixel in image:
    if pixels_seen.hasValue(pixel):
      pixels_seen[pixel] += 1
    else:
      pixels_seen[pixel] = 1

  # how many of the 0xffffff possible colors have been seen
  total_randomness_score = pixels_seen.amount_of_entries / 0xffffff

  # weigh the randomness score 70\% and the stripes score 30\%
  return total_randomness_score * 0.7 + total_stripe_score * 0.3
```