A Genetic Source-Code Program-Synthesizer for Real-Time Co-Evolution with Humans:

Gene-Level Geometric-Push Program-Synthesis

By John Speakman

A Thesis submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy (PhD)

The University of Arts London

Falmouth University

October 2024

Abstract

Can genetic operators be used to produce human-centric source-code documents for human-interactive and collaborative programming practice?

Addressing this question, a novel automated source-code generation system is created which synthesizes, using genetic operators, object-oriented compilable files, which better conforms to human coding conventions than current benchmark genetic algorithms. This project:

- Provides a new form of AI for automatic source-code generation.
- Implements novel use cases for automatic source-code generation in practical collaborative applications.
- Benchmarks the new algorithm against a standard program synthesis benchmark suite.
- Assesses a genetic realignment algorithm in the context of crossover for elitest fitness.
- Explores novel approaches to interaction methods with a genetic auto-coder for multiple simultaneous users.

As of the beginning of this project, there were no existing implementations of source-code generators designed for human legibility while synthesizing for arbitrary language environments: automated program synthesis algorithms derived from genetic operators traditionally operate on specialist languages, with bytecode or machine code solutions or suffered significantly with comprehension against human written code.

The novel algorithm introduced in this thesis interjects into current discourse with a heuristic approach which can operate with relative language agnosticism, while retaining common coding conventions, indentation, arbitrary code length, looping operators, multiple function definitions and function calls.

Using this new algorithm, this thesis explores three diverse experimental phases to analyse potential use-cases, from a qualitative perspective:

An automatic programmer for coding problems: as a tool to support human software developers, for simple programming tasks.

A co-creative environment with Artificial Life: demonstrating the automatic programmer's ability to create evolvable behaviour controllers which adapt to survive under various environmental pressures and to co-evolve with human interactors.

A public facing music generator: as a collaborative medium for live performance environments, providing human-guided fitness training for live evolution of audio.

The thesis concludes that the algorithm is successful in automatic coding for implicit fitness or regression environments, with several limitations relating to the fitness function used and the size of the search space. These limitations are established and persist across genetic methods for automatic coding. Regardless of the limitations, this approach demonstrates valuable use cases in artistic mediums.

Contents

List of Figures

List of Tables

Abbreviations and Acronyms

AFSM	- Augmented Finite State Machine
AI	- Artificial Intelligence
AL, A-Life	- Artificial Life
API	- Application Programming Interface
AR	- Augmented Reality
BNF	- Backus Nuar Form
BOID	- Craig Reynolds flocking algorithm
CPU	- Central Processing Unit
DLL	- Dynamically Linked Library
DNA	- DeoxyriboNucleic acid
GA	- Genetic Algorithm
GADS	- Genetic Algorithm for Developing Software
GE	- Grammatical Evolution
GP	- Genetic Programming
GLGPPS	- Gene-Level Geometric-Push Program-Synthesis
GPU	- Graphics processing Unit
IL2CPP	- Intermediate Language to C++ conversion)
LTS	- Long-Term Support
MR	- Mixed Reality
MVC	- Model-View-Controller
RAM	- Random Access Memory
TBE	- Template-Based Evolution

Publications Arising from this research

J. Speakman, “Evolving Source Code: Object Oriented Genetic Programming in .NET Core,” presented at the 10th Society for the Study of Artificial Intelligence and Simulation of Behaviour Convention, Apr. 2019, pp. 16–19. Accessed: Feb. 10, 2020. [Online]. Available: http://aisb2019.falmouthgamesacademy.com/wp-content/uploads/2019/04/AISB-AI-AND-Games2019_proceedings.pdf

Lorway, Norah, Edward Powley, Arthur Wilson, John A. Speakman, and Matthew Jarvis. ‘Autopia: An AI Collaborator for Live Coding Music Performances’. University of Limerick, Ireland, 2019. Available: https://researchonline.rca.ac.uk/4223/1/AutopiaICLC2020.pdf

N. Lorway, M. Jarvis, A. Wilson, E. Powley, and J. Speakman, “Autopia: An AI Collaborator for Gamified Live Coding Music Performances,” presented at the Society for the Study of Artificial Intelligence and Simulation of Behaviour Convention, Falmouth University, Apr. 2019, pp. 1–4. Accessed: Feb. 10, 2020. [Online]. Available: http://aisb2019.falmouthgamesacademy.com/wp-content/uploads/2019/04/AISB-AI-AND-Games2019_proceedings.pdf

Chapter 1. Introduction

"We recommend researchers to focus more on increasing the readability and interpretability of the generated programs while preserving GP's unique ability to find diverse, novel, and innovative solutions. Furthermore, the community should focus more on providing fast and accessible tools and frameworks to make it easier for researchers as well as software developers to use GP-based approaches"

Dominik Sobania et al, A Comparison of Large Language Models and

Genetic Programming for Program Synthesis [1]

1.1: Rationale and Background

There are very few automatic human-facing source-code generators which can construct multi-line, complete functions, particularly with indentation, loops, conditional statements, and function calls for use in arbitrary language environments. This thesis introduces a new heuristic algorithm which addresses this gap in literature.

This algorithm implements evolutionary procedures to synthesize working solutions to disparate use cases of programs. A major contribution of this algorithm over classical Grammar-based automatic programming implementations is the structuring mechanisms to permit dynamic variable allocation and code blocks with indentation.

This project provides a series of modifications to genetic program synthesis which adopt elements of automatic code generation while following linearizable representations to construct source code, opposed to more common tree-based representations, allowing more direct implementation of biologically inspired correction and optimisation routines and constructing line-by-line code solutions closer matching human readable formats.

The algorithm created, Gene-Level Geometric-Push Program-Synthesis (GLGPPS), appeared to have much broader applications, which have been analysed as a series of human-lead co-evolutionary practices. The results suggested direct and indirect exposure of the algorithm to humans to be utilitarian but limited by the genetic operators that construct it.

GLGPPS is a genetic programming algorithm that automates source-code synthesis by mapping a two-dimensional array of integers to object-oriented code via a scaffold-based templating system. In this design, each gene is translated into a complete code line, and genetic operators—crossover, mutation, insertion, and removal—are applied at the gene level to preserve modularity and syntactic coherence, with a geometric push mechanism modulating mutation intensity based on code indentation.

The algorithm constructed with this thesis does not look to surpass compilation speed, accuracy or completion rates of existing algorithms, instead looks to offer an alternative algorithm for converting a genetic sequence into code. This algorithm provides comparable baseline (before genetic optimisations) performance to current genetic program synthesis models, relative language agnosticism and improved program comprehension.

1.2: Thesis contributions

The contributions of this thesis can be summed up as follows:

1. Provides a series of use-case-dependant variations of new form of human-centric automatic source-code generation, capable of successfully synthesizing code with relative language agnosticism:

    - Gene-Level Geometric-Push Program-Synthesis (GLGPPS),
    - Template-Based GLGPPS (TB- GLGPPS)
    - Needleman-Wunsch GLGPPS (NW- GLGPPS)

2. Demonstrates GLGPPS's ability to generate new, engaging contributions to professional human workflow with limited guidance and small agent population sizes. The findings give evidence that source code generators can be integrated, live, into working environments with beneficial results:

    - GLGPPS provides solutions with lower measured cognitive complexity than the current benchmark solutions for evolutionary program synthesis.
    - GLGPPS can synthesize code for multiple languages: The C# server environment successfully demonstrated the generation of SuperCollider code, demonstrating compatibility with different paradigms and programming languages.
    - GLGPPS does not require reflection to operate: GLGPPS does utilise reflection to execute natively within C# but does not require explicit reflection calls when writing for external environments. This algorithm, when written in languages which are interpreted, should also not require reflection, though this implementation is yet to be conducted.
    - GLGPPS can converge fast enough to be compatible with a non-specialist human audience when applied under suitable circumstances.
    - GLGPPS can be used as an augmentation to alongside human programmers, by modifying existing code in a live coding environment, with some limitations.

3. Provides three series of experiments which assess the algorithm under different conditions, separated into three major case studies with significantly varied requirements, inputs and outputs:

    - Program synthesis given known input and output modelling demonstrating explicit fitness.
    - Evolutionary agents demonstrating implicit fitness.
    - Human collaborative source code generation in a working use-case.

4. Explores genetic, automatic source-coders as an alternative to large language models in program synthesis challenges.

1.3: Thesis Structure

Chapter 2 (Literature review) opens with brief definition of program synthesis and analysing the current dialogue of human-centric program synthesis.

Following this is a primer for evolutionary systems and terminology with biological parallels are utilised to draw observational analogues against the behaviours of the simulated evolutionary models that are utilised in the artefact.

A section is dedicated to exploring a chronological review of seminal developments within simulated evolution, constructing a timeline of papers and analysing their contributions towards developing the field, to contextualise the approach taken to the algorithm designed within this thesis.

This chapter then analyses why evolutionary auto-coders have so far failed to create non-trivial programs, and an analysis of contemporary algorithms with those issues in mind.

Chapter 3 (Research Methodology) is a brief methodological framing, exploring GLGPPS as a terminological breakdown, to outline broadly what the algorithm is then framing the experimental approach taken in this thesis. This provides an overview as to why a mixed-methods approach is taken and broader utility of the research.

Chapter 4 (Artefact Design and Implementation) presents the artefact and its functionality, starting with the core artificial intelligence system, the Gene-Level Geometric-Push Program-Synthesis (GLGPPS) algorithm. This chapter explore the design objectives, theory, and construction of this algorithm, converting a 2D genetic sequence into source code using a scaffolding system, and an example implementation into a C#.net server environment.

This chapter then looks at the implementation of the algorithm into a server, creating the first genetic-programming source-code generation algorithm for runtime C# .net environments. This explores how the system operates and how the genetic operators are implemented.

Chapter 5 (Evaluation) is split into three major components for three distinct series of experiments:

After an analysis of the algorithm's ability to solve trivial coding problems, with classic hill-climbing and a simple unit-test harness for addition, a program synthesis benchmark suit is explored, and a series of comprehension metrics are analysed from generated code.

The second major series of experiments execute the artefact under different conditions and analyse the behaviours of an Artificial Life virtual species and the associated generated code of these agents in these varying configurations. These experiments contrast the GLGPPS algorithm with and without a Template-Based architecture, by observing several longitudinal experiments and exploring a limited amount of human interaction with these algorithms to explore how humans interact with the AI and the how the AI interacts with humans.

The final experiment modifies the algorithm to operate with music generating software, receiving, and outputting data across a network to allow the server to train against real-time human feedback. This demonstrates successful human-focussed training with distinct evolution of sounds with positive reactions from a human audience while demonstrating the use of this algorithm in a time and comprehension critical task with human programmers.

Chapter 6 (Conclusions) concludes the thesis, identifying specific contributions, findings, and limitations of each experiment. This ends with an exploration into potential future work, primarily outlining potential improvements to the core algorithm with an emphasis on reducing the impact of the natural limitations of genetic operators though biologically inspired methods.

Chapter 2. Literature Review

2.1: Program Synthesis

Program Synthesis is a broad term encapsulating *all* methods of automatically constructing code to solve programming problems. This thesis focuses on Genetic Programming (GP) approaches, but this is not the exclusive model in modern discourse.

Program Synthesis has a range of use cases and benefits: discovering novel solutions to problems, helping programmers learn new languages and APIs, evaluating the impact of an API/language change, improving existing code and encouraging creativity in learners.

Two major conceptual models currently dominate the program synthesis domain: models derived from the Genetic Programming (GP) field and models derived from the Large Language Model (LLM) field. In the current discourse, LLM models have demonstrated substantially more interest and growing success in Program Synthesis. Definitions for GP and LLM systems are explored in sections 2.3: *Genetic Programming, Automatically Defined Functions & Object-Oriented Genetic programming* and 2.3: *Contemporary Algorithms*.

Sobania et al. [1] provides benchmarks to compare GP (Specifically Spector's PushGP [2]) and LLM (Specifically Copilot) models, identifying the ongoing strengths and weaknesses of the two approaches. It broadly appears that current comparisons of GitHub Copilot (Large Language Model) and GP (Genetic programming) can comparably solve benchmark problems[1].

Successful solutions between GP and LLM models are not always for the same problems, often resolving with substantially varying length, performance and readability, but this analysis indicates that the two systems are currently comparable for general program synthesis with independent, parallel research in either field. This indicates a level of significance to research in this field.

LLM's utilise large libraries of pre-trained data and often require comparatively very large memory and processor overhead [1] to generate outputs (with current recommendations of 256GB Storage, 16GB RAM and a dedicated Neural Processing Unit [3]).

GP-based systems can operate with very low overheads for similar search requirements in lower complexity searches [1], as they do not require substantial memory and processing [4], searches of limited size and functionality can complete with comparatively very low overhead.

Contemporary GP models demonstrate success in finding novel solutions to problems, with comparable success to modern LLM models. As LLM's operate with data sets derived from human constructed codebases, their solutions are more likely to derive outputs which closer represent stylistically human like programs. This is due to the grammar 'learning' methods of the Generative Pre-trained Transformers [5]. Conversely,

Grammatical Evolution generally derives grammars using a Backus–Naur form (BNF) [6], producing solutions which are generally less readable.

The BNF mechanism can produce complex conjugations of existing code: it is not guided using human-guided use of the discipline language and is subject to generating complex cascades in output statements and generates very high ratios of non-functional code.

Notably, code generated by Copilot is usually easy to read and less bloated than G3P [1]. The forms of BNF conventionally used for code construction in contemporary GP solutions produces complex, hard to read output which does not follow modern coding conventions for most languages. As BNF can contain rules which produce single line commands which are code-length-unconstrained or follow construction methods which may recursively construct within a single line, the algorithm is subject to generate outputs which do not support good conventions for human readability.

To demonstrate this, Sobania demonstrates a pair of example outputs for a 'scrabble score' benchmark of GPT (Figure 1) and GP (Figure 2) algorithms – *"It is noticeable that GitHub Copilot's code is much better structured than the code generated by the grammar-guided GP, which is hard to read and bloated. However, it can be assumed that GitHub Copilot has knowledge about the Scrabble scores of individual letters due to the pre-training of its underlying model."*

```
def myfunc(str1: str):
    str1 = str1.upper()
    sum = 0
    for i in str1:
        if i == 'A' or i == 'E' or i == 'I' or i == 'L' or i == 'N' or i == 'O' \
            or i == 'R' or i == 'S' or i == 'T' or i == 'U':
            sum += 1
        elif i == 'D' or i == 'G':
            sum += 2
        elif i == 'B' or i == 'C' or i == 'M' or i == 'P':
            sum += 3
        elif i == 'F' or i == 'H' or i == 'V' or i == 'W' or i == 'Y':
            sum += 4
        elif i == 'K':
            sum += 5
        elif i == 'J' or i == 'X':
            sum += 8
        elif i == 'Q' or i == 'Z':
            sum += 10
        else:
            sum += 0
    return sum
```

Figure 1: A code example produced by GitHub Copilot for "SCRABBLE SCORE" benchmark, image source: Sobania et al. [1]

```
def tempfunction(in0):
    i0 = int(); i1 = int(); i2 = int()
    b0 = bool(); b1 = bool(); b2 = bool()
    s0 = str(); s1 = str(); s2 = str()
    li0 = []; li1 = []; li2 = []
    res0 = int()
    i1 -= min(mod(i1,(res0 - divInt(getIndexIntList(li0, max(i2, i2)),abs((i2
        * i0 ))) )), max((abs(abs(max(i1, i0))) - (divInt(getIndexIntList(li0,
            res0),max(res0, i1)) - (divInt(i2,res0) * min(i0, res0) ) ) ),
                len(list(map(lambda x: mod(x,max(i0, i0)), list(map(lambda x:
                    divInt(x,res0), li1 )))))))
    for i1 in list(map(lambda x: saveOrd(x), (getCharFromString(saveChr(saveOrd(s1)),
        i2).strip(s0.strip(in0).strip().upper()).rstrip() + (s2 + in0).lower().rstrip()
            .strip().lstrip()).strip(getCharFromString(s1, sum(list(map(lambda x:
                (x + i1), list(map(lambda x: len(x), s2))))[:divInt((res0 * res0),
                    sum(scrabblescore))]))).rstrip(getCharFromString(saveChr(res0)
                        .rstrip().capitalize(), max(max(getIndexIntList(scrabblescore,
                            i2), sum(li2)), getIndexIntList(li0, getIndexIntList(li2,
                                i1)))).strip().lower().rstrip()))):
        if abs(max(max(sum(li1), (i1 * i2)), min(abs(i1), getIndexIntList(li0,
            i0)))) not in scrabblescore[(res0 * int(8.0)):]:
            li1.insert(max(i0, res0),+i0)
        res0 += max(len(getCharFromString(s0, i0).strip(s0)), max(getIndexIntList(
            scrabblescore, i1), min(len(scrabblescore), len(s1))))
    return res0
```

Figure 2: A code example produced by grammar-guided GP for "SCRABBLE SCORE" benchmark, image source: Sobania et al. [1]

This introduces a clear need for change in the construction paradigm for genetic systems for human readable coding conventions in both synchronous and asynchronous systems.

Asynchronous programs run until completion with no further human input. A specification is set before start-up and the program runs autonomously. A common use of asynchronous systems are in pre-defined input and outputs, more commonly addressed in GP systems and demonstrated in the current benchmarking framework for program synthesis [7].

Synchronous, "Human-in-the-loop" or "interactive" programs, allow human interaction to guide the algorithm at runtime with continuous human feedback. They are broad in application and method, generally updating requirements or utilising human preference against example outputs.

2.2: Human-Centric Program Synthesis

Crichton [8] identifies a vision for program synthesis, advocating a shift in program synthesis research from traditional input/output example–based methods toward a human-centric paradigm. The argument is that synthesis tools should not merely be viewed as mechanisms for generating code from explicit examples but should instead function as interactive aids that support programmers. The paper proposes a synthesis framework that emphasizes usability, interpretability, and integration with everyday programming workflows, bridging the gap between automated synthesis and practical software development challenges.

Output code from program synthesis algorithms, especially derived using genetic operators, are generally designed to solve a problem, not to be understood by humans. This is useful in a range of scenarios where a human does not need to know, or the problem cannot be derived by a programmer in a similar time frame, given time for

comprehension, but is rarely useful for a programmer. Programmers classically spend "*58 percent of their time on program comprehension*" [9] without program synthesis. The ramifications of this are extended time spent inspecting, comprehending and modifying complex solutions to coding problems [10], a significant barrier for contemporary Genetic source code synthesis algorithms.

Automatically Measured Maintainability

One approach to measuring conceptual comprehension in code is to analyse the complexity.

A common measure, seen in some example benchmarks of program synthesis [1], is the cyclomatic complexity of an algorithm. This is a measure of the amount of decision logic in a source code function [11]. While this does not directly assess human legibility or conceptual understanding of generated code, it does provide a measure of complexity which contributes to human readability.

$$Cyclomatic\ Complexity = E - N + 2P$$

E = the number of edges in the control flow graph
N = the number of nodes in the control flow graph
P = the number of connected components

Halstead Volume [12] is another metric, which measures the number of distinct operators against number of occurrences of operators.

Both Halstead Volume (HV), Lines of Code (LoC) and cyclomatic complexity (CC) can be used to calculate a Maintainability Index (MI) [13]. The maintainability index utilised in this thesis is the Visual Studio Maintainability index range [14], which uses the formula:

$$MI = MAX\left(0, \frac{(171 - 5.20 * \ln(HV) - 0.23 * (CC) - 16.20 * \ln(LoC)) * 100}{171}\right)$$

This maps into the following ranges:

Highly Maintainable	$=> 20$
Moderately Maintainable	$=> 10 \&\& < 20$
Difficult to Maintain	< 10

With an outline of program synthesis and the identification of a lack of higher interpretability, human-centric program synthesis algorithms which derive from genetic operators, we can explore a novel framework for creating an alternative algorithm.

2.3: Simulated Genetic Systems

This project explores co-evolutionary emergences with humans: a participant may adapt to changes in the environment and perceptual changes in the behaviour of virtual agents. Human interactors may also modify the environment in which the species is evolving, creating a feedback loop incorporating the evolutionary process of the simulated agents, in which participants observe changes in behaviour and the impact of their own modifications and adapt over time to the behavioural changes from the simulated species.

The process of developing a biologically inspired algorithm draws from a theoretical grounding in biological study. This section selectively explores contextually relevant simulated biological processes to construct an argument for the inclusion and development of evolutionary techniques from biological genetic systems, as a primer for a chronological review of simulated genetic systems.

It is perhaps useful to remember that the limitations of biology and the limitations of software are not the same. We cannot necessarily anticipate a direct correlation between natural and simulated systems. We can however anticipate some level of correlation between two mathematical models exploring loosely correlated search spaces. We can identify mathematical theory underpinning some evolutionary systems which correlate with some of the simulated evolutionary systems this project explores.

As this project utilises genetic systems which express both a genotype and a phenotype without explicit non-coding pruning as its core evolutionary process, we will be seeing some similarities in the patterns of the genetic sequences and in the emergence of construction methodologies. In this section, we explore both the biological and simulated underpinnings of these crossovers and emergences in the applicable contexts of an evolutionary model which utilises a Genetic Algorithm (GA) as its core method for evolutionary propagation.

This section explores these concepts as well as defining some of the key relevant terminologies utilised in the field of genetic algorithms, as a primer towards Section 2.3: and to draw parallels in genetic observations throughout the experiments.

Non-Coding Sequence and Loss of Function Mutation

Non-coding regions in genetic sequences, frequently referred to as ‘junk DNA’, is a term used to classify regions of genetic code which do not directly encode protein sequences. The latter terminology comes from the assumption that non-coding regions served no function, though explicit utility has since been demonstrated in these sequences. [15], [16]

While there have been arguments in favour of the permittance of high proportions of non-coding regions in simulated genetic systems [17], [18], most traditional systems using the classic genetic algorithm [19] crossover approach tend to collect very large quantities of non-coding values. These values do not frequently serve to benefit a species and are often detrimental to the system due to the increasing memory required to hold and execute these programs. This leads to negative association of non-coding sequences in simulated evolution models, as program efficiency is generally lost as size increases,

often from non-coding regions, generally referred to as ‘bloat’ [20]. In contrast, in biology, the percentage of gene sequences that are non-coding can vary wildly, though traditionally makes up a very large proportion of the genetic sequence [21].

Non-coding regions can hold value in the evolutionary processes for variable scale genetic systems, both simulated and real. These regions are often used as: buffer regions to change the mutation likelihood of high value regions of the gene sequence, can provide a basis for polymorphism which may lead to speciation and can retain functionality from ancestral species as pseudogenes [22].

‘Loss-of-function’, as the name suggests, is a mutation which disables or reduces the functionality of a gene or coding region, leading to the loss of functionality of that region. This may occur for a range of reasons, but most identifiably from a single allele mutation which disables the entire gene.

A subset of non-coding regions are pseudogenes which have been retained after a loss-of-function mutation. These regions, no longer serving towards evolutionary stability, can undergo silent mutation and may be re-activated through further mutation, to restore fossil functionality in the phenotype of the species or introduce new functionality. [23]

Homology and Convergence

Homology is the similarity of species due to a shared genetic ancestry, where identifiable traits shared between separate species occur due to the species sharing a common ancestor where those traits are due to common genetics [24].

This is notably independent to convergent evolution [25], [26] in which multiple species will evolve a similar phenotype even though they do not share a common ancestor. In experiments with comparatively short search spaces, convergent and homologous species will only be identifiable by a direct search of agents’ ancestry or by analysis of non-coding genetic sequences.

In analysing the genes of a species, we can identify homologous sequences, where agents share sequences of their genetics between species, these sequences are orthologous. As this project looks to focus on the behavioural evolution of agents (opposed to morphological evolution), analysis of orthologous genetics will be the primary identification of speciation events.

A fully converged output in Genetic Algorithm (GA)s is a species who have entirely unified their genome to an identical copy between every agent, where convergence is the approach towards this common sequence. Notably, the point of convergence is not necessarily the global optima [26], though methods such as elitism can move agents towards their local optima, which may include the global optima.

Traditionally, in GA’s directly deriving from John Holland’s classical definition [19], we will arrive at a small number of populations of very similar, though not necessarily identical agents, regardless of how many agents we initialise with. This is for a range of reasons, mostly relating to the inability of the traditional algorithm to speciate, not utilising an environment which encourages divergent evolution or applying a methodology for aggregation-based crossover models (such as herding [27]) , which may drift across a global search space rather than sample into a local search space.

Genetic Drift

This is a statistical process where, by selecting a small random sample population from a large population with randomised alleles, there is a high likelihood of deriving a sample which is not directly representative of the original population. This inconsistency usually either gives lower proportions of certain populations than the source, or only has select populations [28], creating a convergence towards the average or mode of the sample, distinct from the original population.

As the new sample may have very small numbers of certain alleles from the original population, the less frequent may be rapidly lost within the gene pool and a new, dominant species may arise. This process can allow a species to either generate a monoculture or found a new sub-species.

Due to one of the proposed experiments in this thesis involving geographically dependant breeding, a system which utilises random mutation while operating under complex and varying external pressures, we can expect to see regions and periods of high and low population densities alongside geographically distributed populations, which will lead to genetic drift. This may occur from both the founder and bottleneck effects [29] as small monoculture populations, for example a single breeding pair, may move into their own geographic region and reproduce or, as the population decreases to a critical state, the population may become so low that random distributions of genetics are lost.

Gene Insertion, removal, and Amplification

The project will explore the simulated application of sequence length modifying insertion and removal. This is the addition or removal of genes in a genetic sequence and directly modifies the order of genes in a sequence. This mirrors biological systems of similar function [30], where insertion mutation occurs, usually because of incorrect crossover processes occurring [31], where a section from one chromosome is injected into the incorrect corresponding chromosome, inserting a section of the prior chromosome into the new chromosome.

This process can drive mutation, directly modifying the length and structure of chromosomes within the genetic sequence. This makes this process critical for variable gene-length dependant search space mutation.

Notably, this process can cause gene amplification [32] – the process of gene duplication, beyond just the initial crossover misalignment. A species which undergoes insertion and removal processes may produce agents with chromosomes of varying length. This varying length causes the point of crossover to decide between a longer and a shorter genetic sequence, which can cause a duplication of genes due to the alignment of a gene in one parent being different from the other, causing the same sequence to be inserted twice into a child [33].

While there are other methods for insertion and removal of a genome [34], [35], they are not applicable to this study.

Punctuated Equilibrium and Criticality

In adaptive models, periods [36] or architectures [37] of criticality frequently form the most significant adaptions from longitudinal studies, arguably to the extent that independently, more linear Darwinian mechanics may be relatively insubstantial.

Punctuated equilibrium is a modification of Darwinian evolution, in which the gradualistic evolutionary process is punctuated by a period of heightened criticality, often a point where the population dramatically reduces in size, becomes isolated or by other means undergoes a period of rapid mutation. This produces a short burst of evolution over a comparatively small time span and, following fossil records, arguably constitutes the majority of the evolutionary process in biological history.

In more modern studies of explicit genetic deviations in biological systems, we can see the major changes in morphological phenotype derived from minor changes in genetic sequences [38]. These large-effect mutations are driven by short alterations of the genotype, which provides a basis for sudden changes in taxa of species, which in turn can create a speciation event – where species deviate enough to become distinct.

A study into the application of punctuated equilibria for artificial life can be seen by Jonnal and Chemero [39], producing models which suggested that artificially induced periods of punctuation consistently improved overall fitness of the measured agents. This model uses a neural network set to explore a simple environment, a grid, which either gains or detriments the agents score depending on the tile, where the environment would consist of a randomly distributed set of tiles with a consistent number of tiles of each type between generations. This uses a finite set of weights which are used to vary their criticality each generation by a random rate. The results indicated a consistent improvement for all experiments against a control sample which did not utilise explicit interjection of punctuation.

Furthermore, a critical state may be formed by the presence of co-evolution, where "*sustained fitness is optimised when landscape ruggedness relative to couplings between landscapes is tuned such that Nash equilibria* [40], [41] *just tenuously form across the ecosystem. In this poised state, co-evolutionary avalanches appear to propagate on all length scales in a power-law distribution. Such avalanches may be related to distribution of small and large extinction events in the record*" [42].

While punctuation could be expected to occur to some extent naturally in dynamic systems at least in terms of variable evolutionary rate due to more complex relationships with a species environment [43], using a model which dynamically modifies the mutation ratios of an artificial species with human agents may allow a more robust adaptation to human interaction. This thesis does not directly explore the utilisation of explicitly enforced mutation ratio modification during runtime, but the artefact and associated implementation is compatible with this approach and is discussed in Future Work: Artificially Enforced Punctuated Equilibrium.

Punctuated equilibria may occur due to human interaction creating sudden, severe changes to species ecological niche, forcing scenarios which may lead to either sudden evolutionary adaption or population die-off. This may, for example, come from a human placing harmful objects in a geographic region a species has evolved to explicitly utilise.

Co-evolution

Co-evolution is a process under which multiple species exert selection pressures on each other, to add or relieve environmental selection pressures. For the purposes of this paper, we will be analysing the impact of co-evolution between evolutionary species emergent from speciation, explicit predation, and human-AL co-evolution.

The process of co-evolution will modify the search space, altering the local optima of species involved over time and frequently encouraging either further adaption into an ecological niche or stimulating the diversification of species. Co-evolution is a primary driver for speciation [44].

Co-evolution as an evolutionary pressure in Artificial Life models is classically demonstrated through predation, demonstrating predator and prey dynamics and the subsequent Lotka-Volterra equation [45], [46]: a generalised formula for predator-prey population dynamics, identifying the growth and collapse of predator and prey species over time. When applied to evolutionary models, this variance in population creates low population regions, which permit entry points for genetic drift in both species. The evolutionary process here tends to move the local optima of the prey species towards predation mitigation strategies, such as increased birth rates, protective morphological changes, or adaption in behavioural strategies while changing the local optima of the predator species towards more effective predation strategies and morphologies.

Parasitism is also a form of co-evolution, where one species is explicitly dependant on another to the detriment of its host. This constructs a similar arms race to predation, though often produces much more nuanced outcomes, as the parasite is often dependant on its host's survival, producing a generally unidirectional dependence [47].

There are also a range of mutualistic and relatively mutualistic interactions, for example the sharing of food resources or the dynamics of multi-species operations [48]. These may be subtle but still act as a driver toward the development of divergent niche exploration. As an example, the behaviour of one species may cause the geographic optima for another species to shift.

This co-evolution is a form of collaborative practice between the interactors and the species itself.

Sequence alignment

Sequence Alignment is *"The procedure of comparing two (pair-wise alignment) DNA or protein sequences by searching for a series of individual characters or character patterns that are in the same order in the sequences."* [49]

Sequence alignment classically expresses into two general pair-wise alignment strategies: global and local alignment (Figure 3).

Global alignment identifies the optimal alignment between both strings, including the removal or addition of alleles, to preserve the highest number of duplicate values between two strings for the entire gene length. Global alignment is subject to misaligning coding regions of a sequence so that a higher ratio of total alignment is matched.

Local alignment focuses on identifying a region which identifies the highest alignment similarity between the two genes but only carries forward those regions.

For the purposes of this thesis, only Global alignment, using a modified implementation of Needleman-Wunsch, is assessed. As global alignment is applied at a gene-by-gene level for genes of limited length, sub-sets of alignment are not yet explored.

Figure 3: Global and local alignment distinction. Source: Mout [49]

Needleman-Wunsch

The classic global sequence alignment algorithm, Needleman-Wunsch [50], finds the longest common subsequence (LCS) that maximizes a scoring function. The LCS may contain gaps (insertions or deletions) to achieve the best alignment.

Complexity approximation of $O(nm)$ for both time and memory, where n and m are the lengths of the analysed sequences. Optimisations of this algorithm exist, but are not implemented in this thesis [51].

The algorithm defines a score for matches, deletions and insertions. This value is used as a multiplier in determining a match score, and therefore the output strings. As an example, a value may be set to zero and have no impact on the calculation.

This algorithm constructs a 2D matrix of length (string S length+1, string T length+1), a representation of characters in the string across each axis. This matrix is populated by values derived according to the distance between aligned values according set scores for the following scenarios:

For each cell ($M[i][j]$), compute the score by considering three possibilities:

Match : $(M[i][j] = M[i-1][j-1] + s(S\_i, T\_j))$

Gap in sequence (S): $(M[i][j] = M[i-1][j] + d)$

Gap in sequence (T): $(M[i][j] = M[i][j-1] + d)$

This algorithm is then traced back from the bottom right of the matrix to the top left, following the highest score of moves at each cell traversed, moving up, left or diagonally left and up. The result is an alignment score and a pair of strings who represent the optimal alignment.

This implementation is explored as a method for error correction and genetic stability in implicit fitness tests in this thesis. Notably, this algorithm is not common practice in genetic algorithms as research classically focuses on phenotypical outputs or bitwise comparators.

2.4: A Chronological Review of Developments Towards Gene-Level Geometric-Push Program-Synthesis

The following section explores a chronological timeline of major seminal developments of Artificial Intelligence systems as part of a literature review towards a new algorithm, following the structure described in Figure 4. We explore this timeline as GLGPPS was not conceived as a direct iteration of a single algorithm but constructs from range of genetic programming literature.

Due to the rapid variation in potential environments brought by human interactors alongside the expectancy of humans to see rapid alterations in behaviour, this concept brings an emphasis on developing an AI controller capable of highly adaptive behaviour.

This has led to the development of a novel evolutionary algorithm, to meet these requirements. Taking concepts of artificial life towards more biologically inspired genetic selection methods while moving towards an automatic programmer, we arrive at a Gene-Level Geometric-Push Program-Synthesis (GLGPPS) algorithm, which builds on an existing history of constituent theories, which this section explores.

This follows a trend of gradual, continuous improvement of evolutionary systems; however, we have not yet developed a computer program capable of automatically generating high level code to solve arbitrary programming tasks, using evolutionary algorithms. Genetic automatic programmers have been a field of research since 1958 but have continuously failed to solve nontrivial coding challenges. This implies an inherent limitation of traditional evolutionary mechanisms being bound to exponential processing requirements against a linear growth in output complexity, leading to an argument against the use of genetic operators for this context, which is covered within this section.

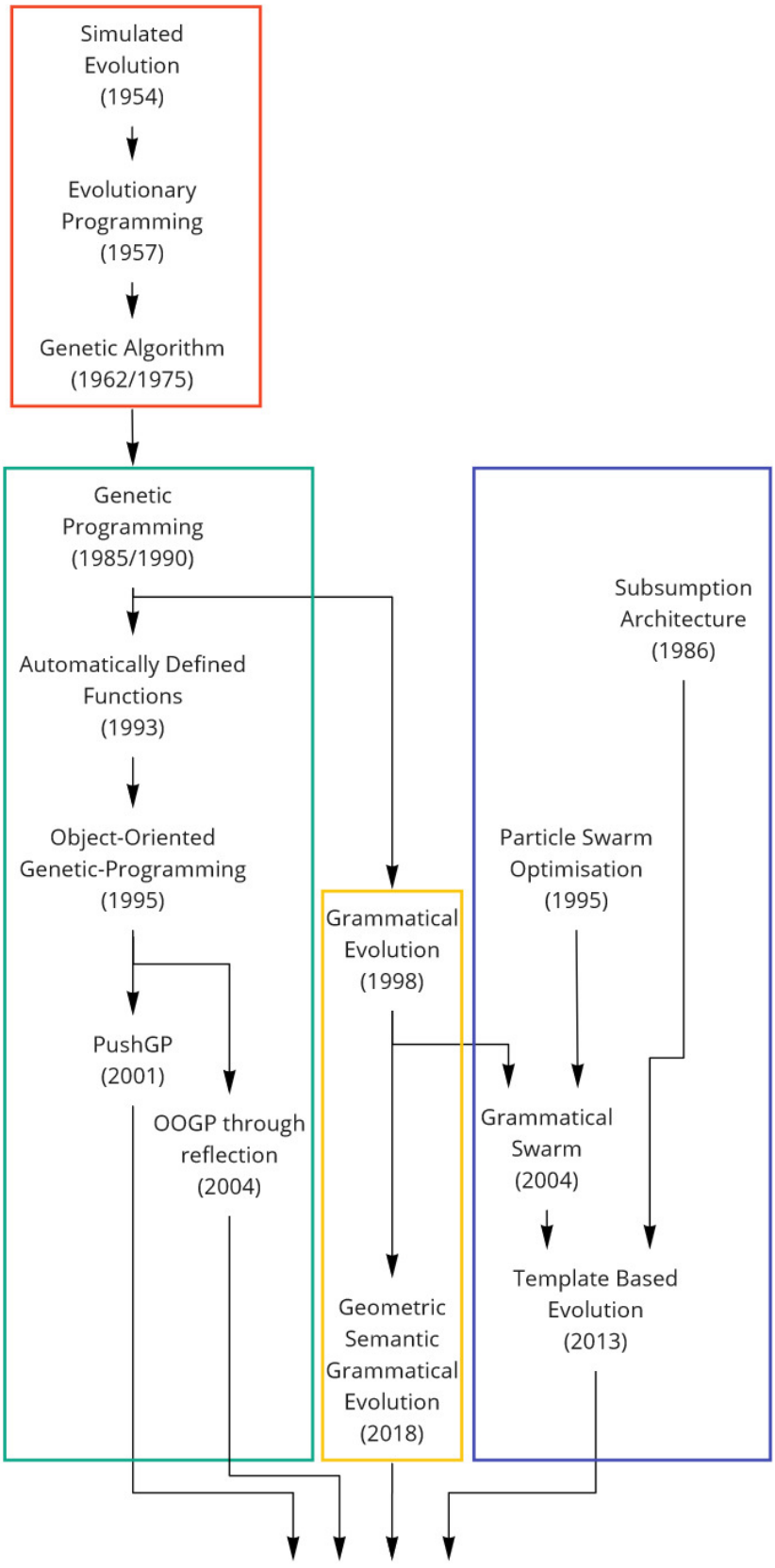


Figure 4: Timeline of seminal development towards a Gene-Level Geometric-Push Program-Synthesis Algorithm

Simulated Evolution, Evolutionary Programming & Genetic Algorithms

In this section, we explore the first recorded simulations of evolution (Figure 5), as a progression of developments building towards the first genetic algorithms.

As these early algorithms mostly fell into exploratory simulation with vague direction, they fall under the broader term 'Simulated Evolution', referring to any attempt to simulate the modification of a phenotype over multiple generations, encompassing genetic algorithms, Genetic programming, Evolution Strategies, and some variants of Artificial Life. Within this, 'Evolutionary Programming' is a term for any algorithm which utilises simulated evolution to automatically develop code.

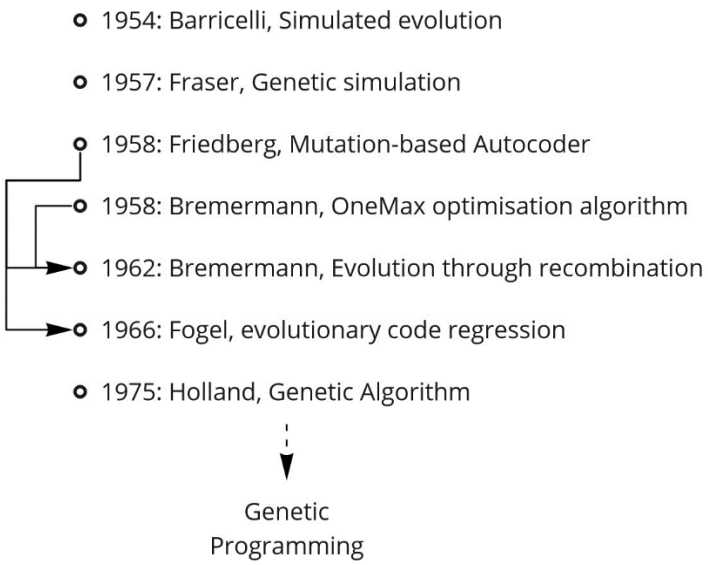


Figure 5: Timeline of seminal developments, simulated evolution to genetic algorithms

The late 1950s saw the first attempts to simulate evolution; this phase of development saw researchers independently creating the first evolutionary software methods [52]. This period has been well researched and documented by Fogel [53], claiming the idea of simulated evolution may have arisen independently, 10 times over two decades.

Contrary to this, John Holland, who claims the development of the original evolutionary algorithm, argues this period "*fared poorly because they followed the emphasis in biological texts of the time and relied on mutation rather than mating to generate new gene combinations*" [19], though this broad dismissal is perhaps a symptom of the lack of awareness of the research undertaken in this period.

The first of these early simulations of evolutionarily inspired algorithms came from Barricelli, in "Symbiogenetic Evolution Processes Realized by Artificial Methods" [54], [55], demonstrates the earliest recorded evolutionary and Artificial Life simulation [56], though the theory for self-replicating automata was well founded by this point [57]. This model demonstrated a study of emergent patterns in deterministic cellular automata, using a matrix of values which vary each generation following a series of rules, in a similar format to the later Conway's Game of Life [58]. Barricelli's simulation took an

initial population as an array of bits and applied a simple ruleset, updating or 'evolving' in defined time-steps, flipping each bit on the following step depending on the state of the surrounding bits.

This produced the first simulated example of evolutionary emergence. Emergence [59], [60], [61] of complex systems (behaviours, patterns) from the interaction of agents with behaviours which do not exhibit those systems independently. "*Emergence occurs in systems which are generated… the whole is more than the sum of the parts in these generated systems. The interactions between the parts are nonlinear, so the overall behaviour cannot be obtained by summarising the behaviours of the isolated components*." [62]

Following parallel research into genetic theory at the time, Fraser, in *Simulation of Genetic Systems by Automatic Digital Computers I* [63], [64] provides the first documented example of a simulation of evolution though the breeding of two simulated gametes using a genetic sequence, represented as a binary string. A phenotypic value was derived from a function run against these strings, as a basis for selection.

This paper took inspiration from the Monte Carlo method [65], [66], using random seeding and random sampling over a complex search space formed from many genes through genetic recombination, and gametes in mating selection. This paper also introduces mutation, initially labelled as "Environmental Effects", as a random deviation of the genetic sequence, independent of crossover. This paper also discussed selection, though a fitness function is not clear.

At this point, the necessary mechanisms which would eventually be labelled "Genetic Algorithms" were present, particularly with Fraser's expanded exploration through the 1960s [64]. Fraser's work during this period, however, appears to have been "*largely ignored by the evolutionary computation community*" [67].

Independent of this research, Friedberg, in "*A Learning Machine: Part I"* [68], developed the first method for evolving working machine code using pointer control. This algorithm relies on a form of guided mutation to evolve, determining the likelihood each gene is detrimental before applying mutation to individual genes.

This system works by moving through an array of 64 pointers which direct to operations, a finite number of pre-assigned operation codes in memory. When called consecutively, these operations will run as a program by pointing at operation instructions in memory and assigning them with values.

This algorithm includes a form of 'GoTo' statement, an operation which redirects the current position of the main operation pointer. This is shown to produce some statements which do not terminate or run for long durations. This in turn led to the introduction of a timer, automatically terminating the program after a set interval. This simple addition can enable the search space to explore recursion and loops without terminating the search itself.

The evolutionary process in this method was a simple form of mutation, modifying a random value or instruction periodically. The algorithm extends this method with a credit-assignment system, analysing the use of every operation in an agent, using the current fitness of the agent against its previous iterations, to determine if each operation appears to beneficial or detrimental, either disabling or swapping the operation if it is shown to cause reduced performance over time. A selective bias is also introduced in this

system, giving both an improved fitness score for agents with a lower number of operations and a simple analysis for the successful completion rate of agents.

Friedberg's model was criticised by Minsky [69] in 1961, arguing Artificial Intelligence models at that time as "*An important example of comparative failure"* since *"The machine did learn to solve some extremely simple problems. But it took of the order of 1000 times longer than pure chance would expect*.". This criticism, while an exaggeration[70], identifies the core issue of the evolutionary approach taken in this model, stemming from the single-target, re-enforcement-mutation approach, the "Mesa Phenomena". This is a form of local-maxima entrapment for hill climbing procedures, "*in which a small change in a parameter usually leads to a small change in performance or to a large change in performance*" and suggests the use of meiosis and genetic crossover as a system to reduce the significance of this entrapment.

A Learning Machine: Part II [71], published in 1959 attempted to improve the performance of this first model. The most significant improvement it delivered was the separation of the problem into individual sub-programs. This appears to effectively form a simple hierarchy, having several base nodes to consistently split significant procedures, effectively forming a simple 2-layer tree. This paper was also reviewed in the same article by Minsky, but it appears that the insight driven in Friedberg's second paper has remained overshadowed by the criticism given by Minsky of the first.

Simultaneously, Bremermann in "The Evolution of Intelligence" [72], began work into a model of mutation-based search space exploration. This introduced the 'OneMax' algorithm, an assessment of the fitness of an agent by comparing the number of correctly allocated bits in a string and brought further insight into the mathematical limitations of this algorithm and the probability of favourable mutations. This heuristic method allowed a comparison of agents and a basis for generational optimisation.

Bremermann later published Optimization Through Evolution and Recombination [73], which took into account Minsky's comments on Friedberg's work. This produced, with results, a new model which took a population of individual agents and discussed a model which could recombine their components between the population, "*by 1962 there was nothing in Bremermann's algorithm that would distinguish it from what later became known as 'genetic algorithms'*."[74]

A notable addition to this timeline is Fogel's work in "Artificial Intelligence Through Simulated Evolution" [75], demonstrating a form of evolutionary regression, in an approach which modifies parameters within a fixed-structure, finite state machine. This method utilised existing algorithms, replacing hard-coded variables with evolvable variables, to some success- but only towards simple parameter optimisation.

Holland expanded this in 1962 in "Outline for a Logical Theory of Adaptive Systems" [76], looking to construct an analysis of the functionality in existing evolutionary algorithms, constructing a theoretical basis for his later work into genetic algorithms, rather than providing a novel algorithm. This paper acknowledges Minsky's analysis [69], though in no great detail and gives no clear reference to the works of Friedberg or Bremermann.

Holland later coined the term 'Genetic Algorithm', in 1975, with the seminal book "*Adaptation in Natural and Artificial Systems"* [77]. This text explores the mathematical theory, with proofs, behind evolutionary algorithms, introducing his own theoretical

framework to this understanding and creating a unified, if not novel, approach to Simulated Evolution in Genetic Algorithms, which spurred the development of the field.

Holland's Genetic Algorithm is a heuristic search algorithm which modifies a representation of possible solutions, or 'agents', over a series of generations, exploring a search space to locate higher fitness solutions, emulating some of the core principles of sexual evolution. This algorithm comprises of the following four steps:

1: Generate a random population of genomes: a sequence of variables (or 'alleles') which impact the functionality of the agent and/or its environment, when implemented in their run-time environment. Generally, every agent in a simulation has its own genome, held independently of all other agents.

2: Assess the fitness of the agents: a score is applied to each candidate solution, where each agent is given an input and its resulting output is used to derive a quantitative measure according to the height achieved within agent's fitness landscape [78]. This may be Explicit or Implicit, either directly deriving a fitness value, and using this value to determine breeding selection, or utilising an environment in which more fit agents are less likely to be eliminated or more likely to breed successfully. A common example of Implicit fitness is in continuous artificial life simulations, where the survivability and agent's adaption to ecological niches acts as a fitness and selection determinant.

3: Breeding between two agents: the next generation of agents are generated through recombination of the previous generation's chromosomes. This process begins with selection, identifying a pair of agents to act as parents, with a higher selection bias towards those with a higher fitness. From the chromosomes of the parents, a new chromosome is generated, using some form of genetic crossover. Each allele in the new chromosome has a probability of mutating, where a mutation will modify a value to a random new value.

4: Removal of agents: agents, with a bias towards those with a lower fitness, are removed from the simulation. This reduces genetic stagnation and prevents the exponential growth of processing requirements. Following this step is a loop back to step 2 unless a condition for termination is met.

Holland's Genetic Algorithm and its explosive popularity set the mechanism for a broad range of derivative applications, including evolutionary programming strategies.

Genetic Programming, Automatically Defined Functions & Object-Oriented Genetic programming

Combining the concepts of Evolutionary Programming and Genetic Algorithms, we arrive at Genetic Programming, explored in this section (Figure 6). Genetic Programming is an extension of John Holland's Genetic Algorithm architecture, when applied to breed populations of potential software solutions, through correlating the agent's alleles with a set of pre-defined functions. The utility of complete program synthesis is explained by Koza:

"The simple reality is that if we are interested in getting computers to solve problems without being explicitly programmed, the structures that we really need are computer programs. Computer programs offer the flexibility to perform operations in a hierarchical way, perform alternative computations conditioned on the outcome of intermediate calculations, perform iterations and recursions, perform computations on variables of many different types, and define intermediate values and subprograms so that they can be subsequently reused." *John Koza* [79]

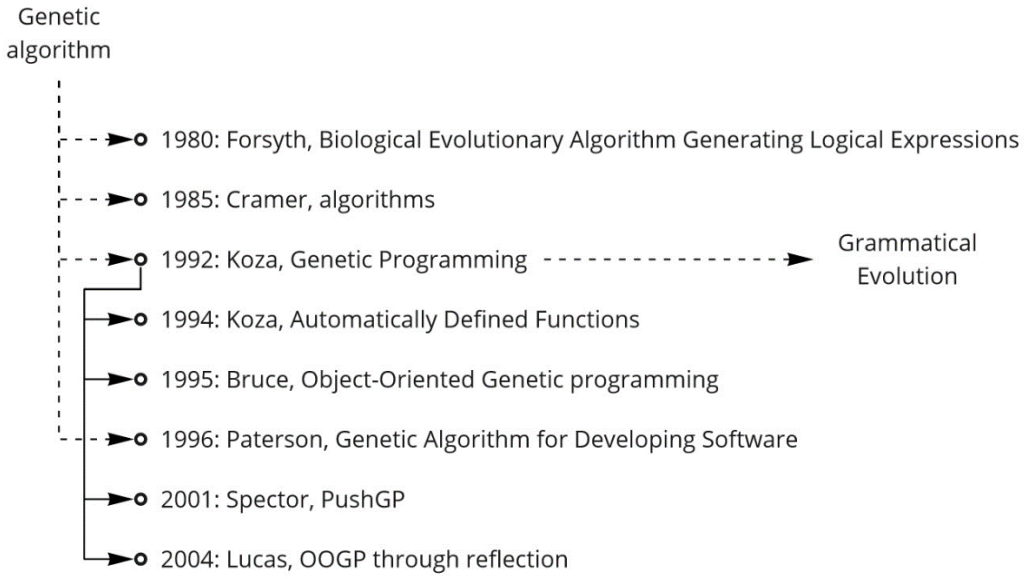


Figure 6: Timeline of seminal developments in genetic programming

Most algorithms within this field revolve around the manipulation of 'abstract syntax trees' (Figure 7), a node-based tree representation of a program's syntax, separating functions and variables as branches and leaves. This syntactic awareness also changes the way recombination and mutation function, as a linear genetic sequence is not interpreted linearly, and the transposition of a branch may modify the functionality of the generated program while maintaining the core behaviour of the agent. Traditionally, in this structure, a leaf would represent a value or variable and a node would represent an operator or function call.

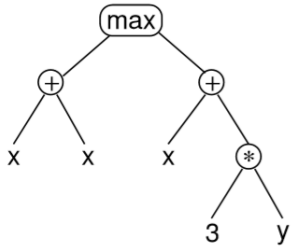


Figure 7: Abstract syntax tree for: Max(x + x, x + 3y)

Source: Adapted from [80]

An element within the representation of the tree may be mutated, swapping values randomly or branches for alternative valid syntax abstracts (for example, mutating a plus into a divide command). A type of mutation specific to tree interpretations may also be applied through the addition of a new sub-tree, where a new, randomly generated syntactic tree structure is appended in place of the leaf node of the child (Figure 8).

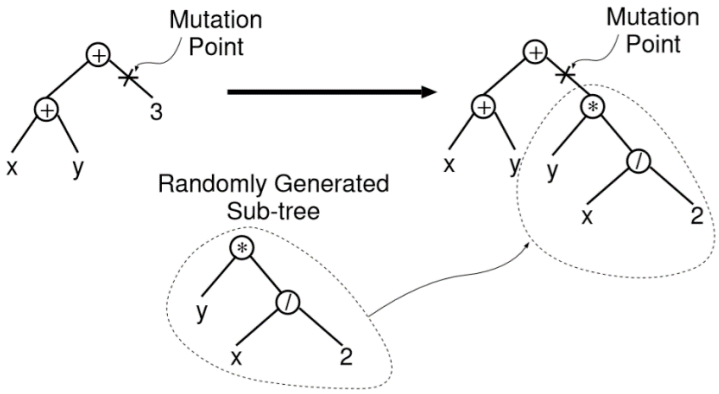


Figure 8: Example of Sub-tree mutation.

Source: Adapted from [80]

The first of this class of algorithm was developed by Forsyth, with BEAGLE [81] (Biological Evolutionary Algorithm Generating Logical Expressions), the first working implementation of evolvable programs using syntax trees. This initial paper produced little impact within the field [82], yet pioneered many of the fundamental mechanics. Of particular note is the use of syntax-aware mutation and recombination on variable-length, tree-structured genotypes, as discussed above, and a simple form of optimisation, in the automatic removal of redundant code (code with no functional impact, e.g. double negatives).

BEAGLE consisted of two programs, a breeding program, and an execution program. The breeding program did not use a classic genetic algorithm at its core, though still evolved executable solutions through sub-tree recombination and mutation, this method retains the highest scoring quarter of agents form the previous generation and hybridises, with mutation, the remaining agents to construct the following generation. The execution program ran the solution formed by the breeding program and gave the output its associated score, to be utilised by the breeder.

This algorithm, as with most classical Evolutionary Programming systems, produced logical expressions rather than Turing complete programs; it is not capable of realising all possible algorithms [83]. This limitation restricts the potential output of these systems, as they cannot explore the entire search space of algorithms – it is incapable of acting as a complete general problem solver.

Cramer [84] independently produced two algorithms: in his first experiment, evolution occurred over a linear integer-value genetic sequence, including crossover and mutation. Here, an integer is a representation of (a pointer to) a syntactic statement or variable, where programs were formed from linking, linearly, a series of statements together according to their position in an ordered memory space, allowing evolution to an array of pointers representing a program to generate novel program solutions. Cramer identified several issues with this approach, most significantly, that slight mutations or alterations to the length of a sequence would eliminate most of the algorithm's beneficial features at breed time.

Cramer's second simulation looked to mitigate the issues raised by this evolutionary approach, stepping away from classic genetic algorithms by again looking towards syntax tree structures. This included the exploration of sub-tree crossover, selecting a branch of one agents' tree and injecting it into the crossover point of another to create a new algorithm. The mutation of agents was limited to the leaves of the syntax trees, due to early experiments indicating that mutations to low depth nodes in syntax trees are generally more disruptive than beneficial.

Koza produced a patent [85] which appears to have rediscovered Cramer's second algorithm. Here, Koza makes an argument for the binary level representation of the genetic algorithm, in support of its benefits to the exploration of nonlinear space, however, he does not explicitly outline a specific crossover or data storage mechanism, making no clear explicit separation between this algorithm and Cramer's revised algorithm, retaining those genetic crossover and mutation mechanisms described by Cramer.

LISP is used due to the syntactical representations of S-expressions which can be interpreted as Abstract Syntax Trees (ASTs) with distinct leaf and internal node objects with symbolic representation. While this approach has proved fruitful in the automatic generation of functional code, it has also moved the general direction of evolutionary programming away from the object-oriented paradigm or human facing syntax and toward the functional representation solution space.

Koza then coined the term 'Genetic Programming' in his book of the same name [79]; this book took a similar role in the AI community to that of John Holland's book for Genetic Algorithms, clearly exploring and articulating the mathematical proofs and applications of the algorithm with a series of seminal works which popularised the field. This was the first book in a series, the following entries [86], [87], [88] highlighting a series of seminal developments within the specialism.

Koza and Rice then expanded the Genetic Programming algorithm with 'Automatically Defined Functions' [86], [89], [90], introducing a new method which calls one evolved function (subroutine) from another evolved function, both of which were automatically evolved through Genetic Programming. This approach allows the independent evolution of multiple distinct functions with a reduced risk of the sudden outbreeding of major elements of the genome, a similar evolutionary system to chromosomes within a genome.

Koza again relies on LISP for this mechanism, as it allows dynamic definition of subroutines which may be called within the same program.

Expanding on this algorithm, Bruce, in his doctoral thesis, developed Object Oriented Genetic Programming [91], [92], moving away from functional and procedural programming paradigms. This took the same syntactic tree representation and evolutionary approach as seen in the previous GP interpretations. Notably, William Langdon simultaneously and independently produced a similar algorithm [93] to evolve abstract data types.

Object Oriented Programming, introduced in 1961 with the Simula languages [94], is a software design paradigm which modifies data structures, the objects, using functions – mathematical transformations and manipulations of the objects [95]. This approach allows for improved modularity and re-use of software, easier utilisation of recursion and reduced complexity of common data modification functions. Simon Lucas gives a thorough exploration into the use of the object oriented paradigm and its benefits in Genetic Programming [96], summarising the exploration with: *"we strongly encourage future applications of GP to avoid the use of functional representation, wherever possible, and operate on an OO space"* [97], a stance which has not been universally adopted.

In an attempt to separate the phenotype and genotype, Paterson developed GADS (Genetic Algorithm for Developing Software) [98], [99], which, using Backus Naur Form (BNF), a grammar which is utilised to create a generic phenotype template, the algorithm was expanded to allow the output to be written in the language of choice, rather than being limited to hard coded frameworks for dynamically generated logical expression compatible languages.

BNF was developed by John Backus in the development of ALGOL [6], a context-free Computational Grammar, a meta-syntax abstraction or representative model, which allows a consistent design for mapping program calls to a standardised format. This generalised representation allows for evolution to occur in a consistent language agnostic data format before being passed to pre-processor which can compile code to match the syntax of the output language.

GADS looked to generate C++ files, requiring a perl pre-processor to generate valid C++ data structures. The syntax used in this algorithm was pre-defined as a list of valid code, which was compiled together using a C++ Genetic Algorithm, using ASTs. This method utilised integers rather than binary, where each value in a genetic sequence would refer to the position of a line of BNF code in the grammar list, which would then be converted to C++.

To expand the capability of GP to multiple data types in a single solution, Spector developed PushGP [2], [100], an extension of the Push programming language, for GP. This algorithm utilises a stack architecture – where objects may be added to the top of an existing data stack with a push or removed with a pop. This push-pop stack constructs code as a string, which can handle multiple different data types, including code itself, by utilising multiple different stacks for each data type.

This algorithm works by progressing through an 'execute' stack, running a series of programs and popping programs as they execute, manipulating the highest objects in the data stacks as they execute, leaving the output as the remaining data in the data stacks at the termination of the execution stack. This demonstrates that a general solution for

solving multi-data type problems may be used in GP, that more complex problems may be addressed, and that GP may yet be applicable to more general software problems.

Outside of languages designed to handle dynamic generation and implementation of logical expressions at runtime, pre-processor applications were necessary for integration of GP with object-oriented languages and were therefore generally not used. To address this, Lucas utilised Java's reflection library to develop Object-Oriented programs through GP. Reflection returns objects, accessible methods, input data type and return type, making it possible for object-oriented languages to discover classes, using a reference to a position in memory, and utilise them at run time.

This approach gave a starting point for full, dynamic integration with pre-existing object-oriented programs, as it may call and be called from other methods within the host program and evolve to fit the demands of the host program from within the host program, applications which following explorations into the use of reflection have elucidated [101-103].

Grammatical Evolution

Ryan and Collins developed Grammatical Evolution (GE) [104], [105] (Figure 9). As with GADS utilising BNF to separate the phenotype and genotype, allowing the manipulation of arbitrary syntax and language.

This algorithm moved away from the manipulation of syntax trees, towards the use of a linear, variable length list of integers. This mapped the Genotype as integers into a BNF grammar, which are mapped onto production rules in the grammar, thereby constructing executable programs. This separation not only renders GE language-agnostic but also enables flexible adaptation of the search space to a wide range of problem domains.

GE has demonstrated strong performance on diverse benchmark problems—including symbolic regression and the artificial ant foraging task—by exploiting its modular encoding scheme. The algorithm's capacity to generate syntactically valid programs from a compact integer representation provides a straightforward mechanism for incorporating domain-specific knowledge through grammar design. As a result, GE offers a significant advantage in terms of extensibility and the incorporation of expert insights without requiring substantial modifications to the underlying evolutionary mechanism. [105-108]

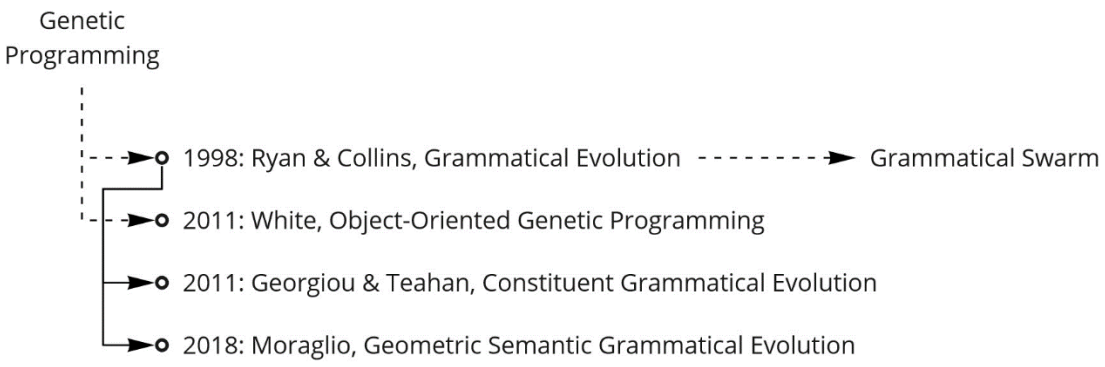


Figure 9: Timeline of seminal developments in grammatical evolution

An immediate issue raised with GE is mirrored with Cramer's first algorithm [84]: due to the linear representation of the code, this algorithm does not have high locality, suffers from destructive crossovers and genome bloat and does not inherit the benefits of modularity from the sub-tree crossover mechanism.

White et al. developed Basic Object-Oriented Genetic-Programming [102], bringing a focus on the significance of operating on objects within the same executable, again utilising Java's reflection library, indicating the same relative language agnosticism as seen with reflection-based approaches in BNF-based AST GP models. This model claimed to differentiate itself from alternative OOGP approaches through the use of a linear representation of genes, through the use of a grammar; while the approach explored in this paper shows to have preferential output against GP approaches, it appears to simply be a re-discovery of Grammatical Evolution (GE) and its successes mirror those seen in GE, though its use of reflection within this search space still holds relevance towards a basis for GLGPPS.

Georgiou and Teahan extended GE with Constituent Grammatical Evolution [109], [110], partitioning the genotype into discrete constituent genes, each encoding a defined segment of the output program and introducing conditional behaviour.

Constituent genes were implemented to minimise the issues brought from a linear code representation through phenotype remapping of grammar. This looked to improve the effectiveness in the number of successful solutions generated, by localizing the application of genetic operators and preserving beneficial substructures during mutation and crossover.

Constituent genes, in this algorithm, takes an approach of defining an additional grammar, supplementing the default atomic grammar. This new grammar is generated by running randomly generated solutions to randomly selected, small sub-problems from the main problem, and assigning the phenotype of the best solutions, directly to the available grammar. This extended grammar is then used as part of the traditional GE algorithm. Immediate issues with this method of constituent genes include: the requirement of pre-existing knowledge of the problem, the reliance on sub-problem selection and an extension in execution time. It does however give an insight to the significance of code-reuse and complex-functionality grammar reintegration.

Conditional behaviour switching introduced if-else statements to the agent's grammar, introducing a method for a functional non-linearity in GE while retaining linear interpretation and evolution strategies, further integration with traditional programming paradigms and greater search space per genetic sequence length.

Experimental benchmarks, including applications to the Santa Fe Trail and Hamilton Court problems, demonstrate that these mechanisms improve both the efficiency of the search and the quality of the evolved solutions

In 2018, Moraglio developed Geometric Semantic Grammatical Evolution [111], a method for deriving the benefits of the geometric layout brought from Genetic Programming into GE. This algorithm broke down code construction into expression trees, developing a secondary sequence attributed to the structure of the representation. In GSGE, solutions are represented indirectly as integer sequences that map to syntactically valid programs via a formal grammar (typically expressed in Backus–Naur Form). Although this indirect encoding facilitates the generation of programs in diverse languages, it often suffers from a low genotype-to-phenotype locality—meaning that

small alterations in the genotype can produce disproportionately disruptive semantic effects.

Moraglio's algorithm brings modularity and a fundamentally different search mechanic to traditional GE, allowing a basis for the identification of high fitness objects automatically during the evolutionary process, rather than on instantiation, as with constituent GE.

Castle and Johnson [112] explore the significance of geometric representation, indicating the impact of mutation and crossover has a greater likelihood of being detrimental at modifications on sections of an algorithm with lower scope, but an equal likelihood of producing beneficial modifications irrespective of depth of scope. The hybridisation of these two concepts suggests an additional optimisation of GE, where mutation and crossover ratios may be set proportionally to scope, based off a geometric semantic representation.

Subsumption Architecture, Particle Swarm Optimisation, Grammatical Herding & Template Based Evolution

This section looks to explore the seminal developments in evolutionary strategies towards Template Based Evolution (Figure 10). This begins with simple adaptive behaviour for early robotics and looks towards alternative evolutionary mechanisms for Grammatical Evolution and the discovery of the principles behind rapid behavioural evolution in a virtual species.

Figure 10: Timeline of seminal developments towards Template Based Evolution

There is an argument that cognition is embodied [113] – that without embodiment, you cannot have cognition. Embodiment, the presence of an interacting body beyond the mind, or cognitive processor, brings the argument that cognition is constructed from the manipulation of the body's environment (including the body itself). This argument suggests that higher level thought is a construction from the abstractions of interaction with the agent's environment. This argument leads to enactivism [114], where "*Enaction proposes to address cognition as the history of structural coupling between an organism and its environment*" [115].

Taking this concept, Brooks developed the 'Subsumption Architecture' [116], [117], an enactive approach to behaviour modelling, looking to develop complex, multi-layered behaviours from simple pre-defined routine modules, for robots. This model takes inputs from sensors and gives outputs as actuators, where the cognitive processes define how actuators are utilised according to the sensors.

This cognitive process is simply a series of simple modular behaviours, in a layered structure. Higher layer behaviours may subsume lower layers – combining multiple behaviours together to form a complex behaviour. Higher level behaviours are those which would normally require the lower level behaviours to also be active to function correctly. This may be implemented as an augmented finite state machine, triggering multiple simultaneous behaviours depending on pre-conditions modified by the input sensors (Figure 11).

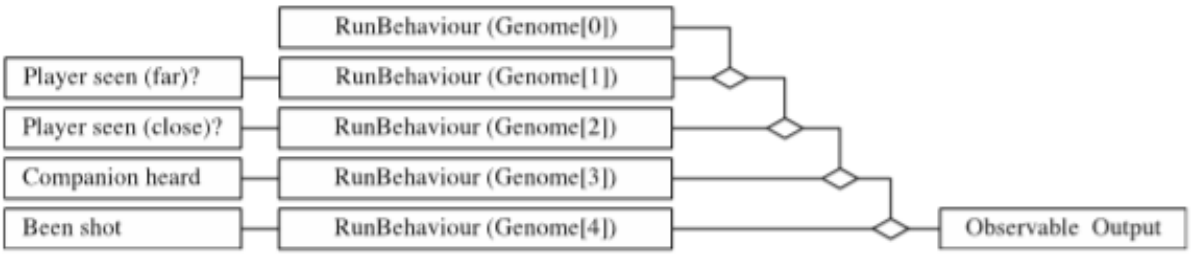


Figure 11: layered subsumption architecture
Source: [118]

Kennedy and Eberhart's seminal paper, 'Particle Swarm Optimization' [119], demonstrates a method for search space optimisation relying on a population (a swarm) of agents (particles) and a fitness function. In this system, agents move towards both their personal best-known fitness and the swarm's best-known fitness, frequently arriving at the maxima of the search space, producing an alternative extrema search method to traditional gradient descent. This system is, itself, arguably embodied, due to agents exploring their search space, as an environment.

O'Neill and Brabazon developed the Grammatical Swarm algorithm [120], successfully merging the Particle Swarm Optimization and Grammatical Evolution models. This approach classified each particle in the simulation as a "*representation of choices of program construction rules, specified as production rules of a Backus-Naur Form grammar*" [121]. This moved the binary string representation of programs in search space towards the agent's independent known extrema and the populations global extrema values, generating novel programs as it moved through search space, usually producing similar results to traditional GE.

Headleand and Teahan introduced Grammatical Herding (GH) in 2012 [27], a grammar-based automatic programming technique that adopts a swarm-inspired approach for candidate solution generation. Unlike Grammatical Evolution, which applies genetic operators such as crossover and mutation directly to a binary genotype, GH employs population-wide herding dynamics to rapidly generate solutions that conform to a Backus–Naur Form grammar. GH operates by treating each candidate solution as an agent whose state is influenced by collective dynamics. The algorithm iteratively adjusts

candidate solutions based on aggregated directional information derived from the population. This mechanism drives a fast convergence toward moderately fit solutions by exploiting local information and group behaviour patterns.

Headleand and Teahan, followed this model with Template Based Evolution (TBE) [122], [123], [124], moving entirely away from GE with a model designed for the evolution of behaviour in situated, embodied agents, focusing on artificial life as a medium for implicit evolution of agent behaviour in multi-agent simulations. This focuses on the behavioural analogues with real world species, stepping back from automatic programming algorithms entirely to instead evolve parameter values for existing design architectures.

This algorithm integrates predefined structural "templates" into the evolutionary process to constrain the search space and guide candidate solution generation. An augmented finite state machine averages behaviours smooths the search trajectory in the behavioural parameter space, effectively creating a "gradient-like" effect without relying on explicit derivative information.

TBE operates by first defining a template that embodies domain-specific coding conventions or problem structure. The evolutionary algorithm then optimizes the variable elements that fill the placeholders in the template. Constraint checking is incorporated during genotype-to-phenotype mapping so that only valid solutions, conforming to the template, are produced. This approach limits the search to a subspace defined by the template, thereby improving convergence speed and solution interpretability compared to unconstrained evolutionary methods. TBE consistently demonstrates notably improved convergence properties relative to traditional Grammatical Evolution, given a heuristic template with suitable methods is applied.

Rather than Fogel's linear architecture, this algorithm is based on a subsumption architecture, a species "template", in which a series of pre-defined behaviours are subsumed depending on an agent's inputs. The value requirements of these triggers and the strength of the outputs of the agent vary using an evolutionary algorithm – for example, if the distance to a target is a switch to trigger, the distance required to activate the trigger may be defined by the agent's genome.

This algorithm looked towards implicit fitness functions, breeding as the agent enters its own breeding period, rather than at distinct generational breeding phases, with the intention of moving towards mating behaviours which are more analogous to nature. Taking inspiration from Grammatical Herding, this algorithm also used physical distance to control breeding, looking towards an Aggregation-Based Crossover Operator, to improve the adoption of favourable genes during crossover.

'Towards Real-Time Behavioural Evolution in Video Games'[118], utilised TBE to develop adaptive agent behaviour for competitive AI controlled characters in a first-person-shooter video game. This architecture allowed for rapid evolution of highly varied behaviour which produced human-like behavioural routines with limited processing overhead.

Yampolskiy's argument against evolutionary auto-coders

Although we have seen many approaches to auto-coder solutions over time using genetic algorithms, we have never converged on an auto-coder that can replace a human programmer in the workplace for non-trivial problems.

Yampolskiy [125], in a 2018 paper indicating the failure of evolutionary approaches to program synthesis, suggests the entire concept of GP is flawed, concluding that "*on close examination, all 'human-competitive' results turn out to be just optimizations, never fully autonomous programming leading to novel software being engineered... Although hill-climbing heuristic–based evolutionary computations are excellent at solving many optimization problems, they fail in the domains of noncontinuous fitness."*

This is echoed by Spector, bringing 'The open Issue' in 2019 [126], to modern programmers, indicating that the field has not achieved a fully functional program synthesis algorithm and suggesting the direction of modern GP has moved away from the original 'holy grail' of the field – an auto-coder which can program like a human, or at least deliver non-trivial programs consistently.

It appears that, as the complexity of the solution space increases linearly, the search increases exponentially. This is not surprising as, to generate more complex solutions in code, you need more lines of code, which for auto-coders means longer genetic sequences. Increasing the length of a genetic sequence increases the number of solutions in the search space exponentially. This means it will take more processing to search that space without either a heuristic guidance proportionally to the increase in dimensionality, or a different approach to search space exploration.

This fundamental law of regression in complexifying space is tied to Moore's law [127], the reason we can explore increasingly complex search spaces over time is due to the computational power increasing exponentially. Yampolskiy points out that Koza's own work [128] identifiably falls into the same limitations.

For the context of this thesis, one note by Yampolskiy is that artificial life created using these evolutionary mechanisms is equally trapped within the same limitations, which we can explore in the findings of our artefact.

It is important to note that Yampolskiy contextualised this paper primarily around Darwinian evolution in the context of a vertical slice of seminal papers. While it is true that a GA which only uses regression may be constrained by this statistical limitation, Yampolskiy does not explore heuristic, modular and non-Darwinian evolutionary models in his argument.

Looking back at Template Based Evolution, while this model did not attempt to produce a program synthesis artefact, the use of templating provides a heuristic mechanism to augment and split the genetic algorithm; in theory this may have had an impact to reduce the search space of the GA. This may be why Headleand and Teahan saw improvements in the evolution of behaviour from Template Based Evolution against prior explorations into GE. As this mechanism is explicitly hard coded in TBE, we can only expect a very limited modification to search space. We will be incorporating this same templating mechanism into the artefact in this thesis to explore the relationship between regression, modularisation, and dimensionality as a basis to provide a theoretical framework to expand beyond the limits of traditional evolutionary models.

Contemporary Algorithms

This section looks at modern and developing algorithms for program synthesis with a similar design, output, or goal to the Gene-Level Geometric-Push Program-Synthesis (GLGPPS) algorithm developed in this project. This analysis has exposed very few source code generators for object-oriented programs, despite the dominance of the object-oriented paradigm in modern programming and identifies alternative, neural-network derived algorithms which successfully generate some level of automatic program synthesis.

Spector's own Push algorithms [2] are still being expanded and implemented across various languages, including within .Net [129]. These algorithms still work on the same fundamental principles as the original PushGP but are periodically revised or expanded [130]. None of Spector's own algorithms attempt to generate source code, though other GE algorithms [131] do. Notably, however, none of these algorithms move outside of those boundaries suggested by Yampolskiy.

DeepCoder by Balog et al. [132], is an extension of an alternative approach to the Genetic Programming paradigm, using Neural Networks (NN) as a top-down, high level framework constructor rather than Genetic Algorithms 'bottom-up construction. This takes an entirely contrasting approach to program synthesis, relying on induction from a trained NN and pre-existing training data sets.

During the development of this thesis, Github Copilot [133] has been released: a currently ongoing development utilising a Generative Pre-Trained Transformer [134]. This model has since become the dominant method in commercial and research use to an unprecedented degree. This model classically demonstrates comparatively superior code comprehension in outputs against GP derived models, including PushGP [1].

locoGP [135] provides one existing model for 'source-code' generation; however the generated representations appear to be lower level than traditional source code, though syntactically valid solutions. This model utilises abstract syntax trees and dynamic compilation of Java bytecode, rather than traditional human-oriented source-code, but does make a strong argument for the plausibility of a source-code generator using dynamic compilation loop systems. This model is particularly pertinent to this thesis as it shares some similarities to approach and aims, though it operates using more traditional GP approaches.

Matej set out with the intention to produce an algorithm which automatically constructed human-readable source code. A following approach, '*Neural Sketch Learning for Conditional Program Generation*' [136], successfully generated short, type-safe code for programs which utilise multiple Java APIs automatically in a contextually appropriate format (Figure 12), notably after Yampolskiv's article.

```
String s;
BufferedReader br;
FileReader fr;
try {
 fr = new FileReader($String);
 br = new BufferedReader(fr);
 while ((s = br.readLine()) != null) {}
 br.close();
} catch (FileNotFoundException _e) {
} catch (IOException _e) {
}
```

(a)

```
String s;
BufferedReader br;
InputStreamReader isr;
try {
 isr = new InputStreamReader($InputStream);
 br = new BufferedReader(isr);
 while ((s = br.readLine()) != null) {}
} catch (IOException _e) {
}
```

(b)

Figure 12: Examples of two automatically generated programs from the same algorithm.

Image Source: [136]

Following on from this work and '*Sketch*'[137] – a programmer guided induction program synthesis algorithm, '*Learning to Infer Program Sketches*' [138] was developed, determining a high level program architecture which leaves gaps for search algorithms to fill in.

Neural networks are effective at high level code structure and replication of trained search results but classically demonstrate poor performance with high computational complexity in smaller search spaces. Solar-Lezama argues that by *"letting the neural nets handle the high-level structure and using a search strategy to fill in the blanks, we can write efficient programs that give the right answer*." [139]

Hybrid approaches, which first architect from the top-down (classification), then construct from the bottom-up (regression), would likely deliver a more successful program synthesis algorithm then either approach independently.

While Yampolskiy's argument [125] appears to still stand for genetic programming systems independently, due to hill climbing optimisation remining trapped with proportional search space requirements against Moore's law [127], Yampolskiy's argument appears to be not applicable to hybrid approaches with top-down guidance. The top-down neural network approach conversely fails to construct algorithms due to a lack of a low-level code constructor or optimization mechanism, opening an exploration into further use of hybrid systems for high level framework construction and low-level optimisation. The success of the early hybrid approaches seen in SketchAdapt [138] and templating in TBE [123] are a testament to this, inferring some value in further exploration into the Genetic Programming paradigms as an element with the intention of developing new hybrid solutions.

The plausibility of successful, fully functional, complex program synthesis is brought by the early results of SketchAdapt suggests exploration in this direction to be timely. With this validation of exploration into the field of evolutionary autocoders hybridised with the concept of the high-level controller, for efficiencies sake in the form of TBE's templates, we can look to the construction of a new algorithm which attempts to push the limitations of genetic operators for evolving source code.

Developing this algorithm into an Artificial Life context allows the exploration of implicit fitness and the correlation and biological emergences from genetic operators

which are better suited for a longitudinal experiment with dynamically changing fitness landscapes. This may prove a more successful, though limiting context to emerge automatic programming but also opens a novel exploration into Artificial Life.

2.5: Summary and Findings of Literature Review

This review has identified several gaps in existing literature which this thesis fills, particularly a lack of source code generation systems stemming from genetic operators, designed to be read and understood by human programmers in conventional use. This is particularly systemic in C# runtime environments and does not appear to have been approached from a single-solution language agnostic framework. It also indicates the general direction of this field of study: towards a universal automatic coding system utilising genetic operators.

Looking at current program synthesis, we identified a significant lack in legibility of genetic systems, a substantial trade-off in utility between GPT-based models and GP based models. This indicates a significant gap in operational models which demonstrates the performance and search methods of GP derived program synthesis with methods for improved human comprehension without significant processing overhead.

Genetic theories from biological sciences offer a grounding framework for the simulation of biological systems. This review suggests states of criticality which may lead to cascading evolutionary events or punctuated equilibrium within a co-evolving ecosystem, raising the significance of permitting dynamic interaction in biological or simulated environments.

Genetic systems within or inspired by biological sciences derive a series of logical operations, some of these operations act as functional elements of the genetic algorithms explored in this thesis. These systems are explicitly programmed as a breed-time operation or data-parsing mechanism. Substantially within this set are gene insertion, removal, and mutation with various strategies for non-coding regions. These elements will direct the construction of the core algorithm in terms explicit functionality of the crossover algorithm itself.

Other genetic behaviours from biological systems occur emergently in the genome of evolved species, such as homology, convergence, and genetic drift. These patterns in genetic data are observed in biology but may emerge due to geographic or behavioural relationships of virtual agents with varying genetic data sets reproducing, subject to the crossover protocol. These patterns and commonalities may be present in a simulated ecosystem under which natural selection with genetic crossover of breeding pairs is present and are therefore utilised as an element of analysis in the qualitative study of the artefact.

Analysing a timeline of genetic systems towards the development of automatic program synthesis algorithm, we can see how the theoretical grounding derived from biology has inspired the development of this field from inception. As time has progressed, cross disciplinary and mathematically abstract theories have hybridised into the field, but genetically inspired solutions to the problem space have never been successful in more than trivial tasks.

This literature review analyses developments towards the artefact's algorithm across multiple converging branches and highlights regions of simultaneous development of functionally similar research. The study indicates that the field itself has been punctuated by seminal developments, with several regions of historic and modern contention. Within the research limitations of this thesis, these generally either increase levels of abstraction, incorporate cross-disciplinary optimisations, or move the search space towards more utilitarian output formats.

Regardless of the approach, evolutionary algorithms have universally failed to produce an automatic software synthesis algorithm for complex problems. An analysis of contemporary dialogue in the field, particularly by Yampolskiy [125], suggests that complex solution space exploration may be mathematically implausible due to the nature of hill-climbing-based algorithms, which includes the genetic algorithm and particle swarm series. Despite this, compounding levels of abstraction, deeper mathematical exploration into biological and statistical optimisation theories and cross-disciplinary techniques may hold some level of value within automatic program synthesis, particularly for shorter length, human-facing toolsets.

With a theory for the design of the core algorithm, the literature review identifies regions within evolutionary inspired program synthesis paradigms may apply. As the algorithm should be capable of demonstrating behavioural adaptation, a context in which this may be effectively demonstrated is conceived: an artificial ecosystem capable of human interaction.

Chapter 3. Research Methodology

Large Language Model (LLM) algorithms such as GPT have experienced rapid adoption due to their usability, readability, and seamless integration into development environments, despite their high computational overhead and concerns related to authorship and novelty. Conversely, genetic program synthesis methods demonstrate efficiency with low to medium search spaces and can generate novel solutions under constrained resources. However, these genetic approaches have low adoption rates, largely due to issues with human readability. Specifically, the prevalent use of recursive BNF structures, unbounded line lengths, and the absence of standardized linting result in code that is difficult to interpret.

This research addresses a critical gap in literature by developing the Gene-Level Geometric-Push Program-Synthesis (GLGPPS) algorithm. GLGPPS incorporates a finite line length "scaffold"-based framework that enforces coding conventions, improving the structure and legibility of the generated source code. The objectives are to determine whether GLGPPS can achieve comparable baseline performance against modern genetic synthesis methods and to enhance the readability and maintainability of automatically generated code for the same solutions.

As this algorithm was designed to support human comprehension without substantial compromise against benchmarks for completion or code complexity, a series of experiments are implemented to test both functionality and human utility. These experiments adopt a mixed-methods approach that integrates quantitative metrics: functionality, accuracy, quantitative comprehension complexity with qualitative assessments of usability and real-time application in environments involving human programmers.

The following experiments are adopted for specific analyses:

OneMax: Evaluates the abstract gene-level operator performance and search efficiency.

Unit Tests: Validate the functionality of the synthesized code against known solutions.

Spector and Helmuth Benchmarks: Provide comparative performance analysis and assess code readability and complexity using established metrics against G3P

Artificial Life: Demonstrates the algorithm's capability in dynamic, interactive environments and tests implicit fitness.

SuperCollider: Validates the utility of GLGPPS in live, human-interactive creative applications.

OneMax

As GLGPPS operates as a gene-level search with specified length alleles, an experiment is run to quantify how this algorithm performs abstractly in terms of fitness of search space without compilation into a phenotype. This experiment isolates the behaviour of the genetic operators in both binary and integer gene representations, converting a random or inverted string into a set solution.

OneMax is used to explore the gene-level search at an abstract level and benchmark abstract complexity scaling and general performance on a simple, well-defined problem. At this level, the utility of a 2D Needleman-Wunsch algorithm for gene alignment as a search is also explored in the context of search space optimisation.

Unit tests

To test the functionality of the algorithm for solving problems, a unit test harness is implemented which can test generated code against either a series of known solutions or a known working solution algorithm.

A common problem in this space is searching for accuracy in irrational numbers, we use Pi as an example benchmark.

For further implementation tests, randomly generated numbers could be passed in large quantity to a working addition algorithm and the solutions for the test algorithm can be compared. A simple sum of three inputs is used, addressing the algorithms' ability to write a simple function which can take three integers and return their sum.

For more complex problems, known solutions are read in from an external file with a known input and an expected output.

Spector and Helmuth Benchmarks

Using the unit test harness, we can look to standard benchmarks for Program Synthesis. A the point of writing this thesis however, the only standard benchmarks are by Spector and Helmuth [7], which advocate analysis to remain at completion only. These experiments will however be analysing the code generated for a range of readability metrics: cognitive complexity, maintainability index, cyclomatic complexity, lines of source code and lines of executable code.

We explore a subset of these benchmarks with a focus on a comparison to known solutions to this benchmark from other genetically inspired solutions, focusing on analysing standard human comprehension metrics to determine if GLGPPS produces more human-centric solutions.

These benchmarks are limited by the number of standard benchmarking tools program synthesis comprehension and the number of existing genetic program synthesis algorithms and benchmarks in the field.

As these benchmarks have been tested on the main contemporary comparison algorithm G3P, results are directly compared for completion, generations to completion and a range of complexity statistics.

Artificial Life

To introduce human interaction, analyse qualitatively GLGPPS in implicit fitness and explore the use of a template-based mechanism to structure evolutionary complexity in a classical use-case, the algorithm is placed into a behaviour controller for a many-agent Artificial Life simulation. This analyses specific emergent behaviours and convergence rates, demonstrating GLGPPS's ability to generate complex, functionally variable code without explicit guidance at a rate compatible with human interactors.

SuperCollider

As GLGPPS is intended for human workflow integration, a model is utilised to demonstrate GLGPPS as a tool in a working environment.

The GLGPPS scaffold library is modified for "supercollider" code, a music language which operated dominantly with mathematical transformations of objects to produce sound waves. This experiment also demonstrates consistent operation to a live audience in multiple languages, including unconventional languages.

The initial experiment demonstrated a live performance with human interactors, providing feedback as a scale: this feedback was used as the fitness function for evolving subsequent generations in real-time.

The second phase of this experiment saw generated solutions directly implemented into musicians' own code as part of a live collaborative performance. The details of this series of experiments can be found in the *Publications Arising from this research*.

Our experiments include assessing abstract gene-level search performance (OneMax), evaluating functionality through a unit test harness, comparing human-centric metrics with other solutions (benchmarks), exploring implicit fitness in artificial life, and investigating GLGPPS within music collaborative practice workflow. These experiments collectively contribute to our understanding of program synthesis and human interaction.

Chapter 4. Artefact Design and Implementation

This chapter details the development of the artefact for this thesis, which comprises the Gene-Level Geometric-Push Program-Synthesis (GLGPPS) algorithm and its integration within a networked server framework designed to host a parrellised, multi-use framework with inheritance of an Artificial Life ecosystem, large file parsing, and mouse and keyboard control for code export.

This describes an algorithm with comparable performance and improved readability to contemporary genetic program synthesis methods, with a platform to apply the algorithm to a range of environments.

4.1: Design Objectives:

Human-Centric Code Generation: Develop a system that synthesizes source code conforming to standard coding conventions (e.g., structured indentation, modular function definitions) to maximize human readability.

Cross-Language Agnosticism: Ensure the algorithm compiles across multiple programming languages by enforcing generic, language-independent formatting rules and leveraging a scaffolding mechanism.

Real-Time Co-Evolution: Support interactive, real-time evolution where the system can co-adapt with human users.

Structured Genetic Representation: Utilize a 2D genetic matrix paired with a scaffold library to enforce a strict, linear construction of code. This approach guarantees that genetic operations (e.g., crossover, mutation) preserve the syntactic and semantic integrity of the generated code.

Geometric and Heuristic Enhancements: Integrate geometric-inspired methods (push-pop stack for indentation control, depth-based mutation and crossover) and heuristic methods to manage complexity and enhance convergence rates during the evolutionary process.

Robust Runtime Architecture: Implement the environment in a high-performance, multithreaded .NET Core server framework that can efficiently manage resource constraints while executing dynamically generated code.

Dynamic Variable and Scope Management: Incorporate mechanisms for automated variable assignment, dynamic scope resolution, and type control to ensure that generated programs are functionally complete and free of semantic errors, mitigating compilation and runtime catch expense.

Optimized Trade-Off Between Efficiency and Flexibility: Strike a balance between evolutionary overhead and the runtime efficiency, ensuring that the system remains responsive during live interactions and under heavy background computational load.

Empirical Benchmarking and Validation: Expose variables for comparison to program synthesis benchmarks, incorporating both quantitative performance metrics and qualitative assessments of code readability and maintainability.

These objectives collectively underpin the development of an innovative, human-centric genetic program synthesis system while addressing existing limitations in automated source code generation.

In investigating the mathematical foundations of evolutionary processes, modifications and extensions to genetic algorithms can be conceptualized as balancing three fundamental trade-offs: adaptive potential, efficiency, and speed of adaption (see Figure 13). An increase in the mutation ratio accelerates exploration of the search space—thus enhancing speed of adaption but concomitantly reduces the probability of reliably converging on a global optimum, thereby diminishing adaptive potential.

Incorporating a templating mechanism to restrict the search space can limit adaptive potential if the global optimum does not lie within the predefined template boundaries. To mitigate these conflicts, the proposed algorithm segregates the evolutionary process from runtime constraints. This decoupling reduces the pressure to balance efficiency against evolution, while still enforcing an upper bound on computational cost to ensure that system performance is not compromised.

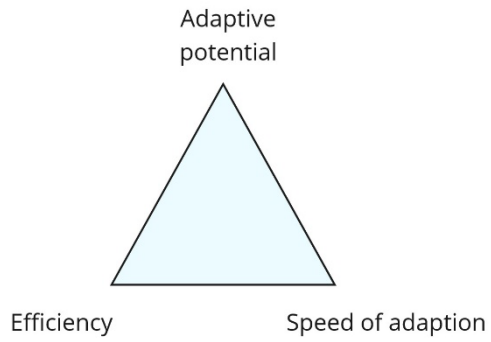


Figure 13: Evolutionary Algorithms Ternary Plot

By separating the evolutionary process from the runtime requirements, we can also reduce the need to balance for efficiency. However, if the efficiency cost is too significant, it will severely impact the performance of the host device and may cause lengthy processing delay, which would detriment human engagement or lead to software failure at hardware capacity when breeding in an AL simulation, so a maximum still needs to be formed.

Accounting for the need to co-evolve with humans, who may have wildly varying expectancies and patience, especially in a system which requires dedicated human interaction [115], the focus on speed of adaption becomes more significant, but so does the complexity of outcomes. The system would need to cater for both dedicated and bursty interaction. The latter process is addressed from a structing method TBE resolved, the prior is addressed through a new heuristic program synthesis architecture, capable of heuristically exploring the entire search space of a programming language.

4.2: A Gene-Level Geometric-Push Program-Synthesis (GLGPPS) algorithm for Source Code Generation

This section covers the development of a novel algorithm at the core of this project:

Gene-Level Geometric-Push Program-Synthesis (GLGPPS), a novel evolutionary process for complex behavioural evolution, with minimal hard coded architectures, for runtime operation in large-scale, multi-agent, artificial-life systems with human interactors. This brought a preference for an algorithm which is as lightweight as possible at run time, would run on modern servers and could be integrated as efficiently as possible into complex applications, with live dynamic evolution.

GLGPPS is a novel approach to program synthesis that enforces a linear, line-by-line construction of source code. By implementing a restrained interpretation of Backus–Naur Form to generate code stubs, the model achieves closer adherence to established clean coding principles compared to traditional genetic programming techniques.

The naming convention is constructed as follows:

Program-Synthesis: The automatic synthesis of software solutions to programming problems. In this case, complete files designed to be executed at run-time autonomously.

Gene-Level: this algorithm operates by constructing a single line of code from a single row of a 2D matrix. For stability, at crossover, the algorithm shares one entire line of this matrix, a whole ‘gene’, at a time. A templating mechanism, which draws a code “scaffold” from a library of pre-defined, human-defined code “scaffold”, is used as the broad outline of a line of code. This outline is then filled out using values from the rest of the gene.

Geometric-Push: using a push-pop stack, a method for constructing algorithms consistently with indentation depth that can handle variable scope control, allows scope control and indentation, with “if”, “else” and “for” loop functionality. This indentation depth can be utilised as an exposed semantic depth value.

4.3: GLGPPS Source Code Interpreter for Program Synthesis

As a source code synthesizer, the algorithm generates lines of code which populate the body of functions. Like the contemporary algorithm Grammar-Guided Genetic-Programming (G3P) [168], this body of functions is placed inside a hard-coded wrapper, which constructs the import statements for required libraries, generates a function header with the proper signature and access modifiers, and instantiates necessary support classes. It also embeds hard-coded variables and control values to manage runtime behaviour. This structured approach ensures that the final output conforms to established coding conventions and is immediately compilable and executable. The main class is also wrapped in a try-catch block, in case of execution failure. Timeout may be applied when calling the class as an async operation from the target application, rather than inside the generated code itself, allowing external control over premature termination.

This process begins with a list of lists of strings, these strings are decompiled lines of code, held within the internal lists, which are used to interpret the genetic sequence into source code.

Gene level Scaffolds

At the core of this algorithm is the gene-level scaffolding mechanism. This scaffold controls the construction of a single line of code. A code block can therefore be constructed with a series of scaffolds. This mechanism encourages atomisation and emphasizes evolutionary pressure towards code structure. Complex redundant structuring can then be more easily optimised with a simple linter.

The scaffolding system uses an integer-based genetic algorithm to control evolution, using a 2D matrix of integer values rather than a 1D array of Booleans. To generate code, the first value in each line in the matrix acts as a pointer to a code scaffold – a line of code which has been separated into component parts, with its variables removed.

Figure 14 illustrates the fundamental concept behind the templating system. A string scaffold is constructed in a fixed pattern: it consists of alternating literal string segments and placeholder markers. These markers control how the use of a variable in this location is handled, either creating a new variable, using an existing variable or reading the value of the gene directly as a value. The length of this scaffold is proportionate to the maximum number of evolved variables required in a library; it is the width of the 2D matrix of agents' genes.

At the end of this scaffold, a pair of integers is appended. These integers serve as control flags:

- The **first integer** indicates the depth flag, which determines how the generated code should be indented—effectively controlling the scope and hierarchical structure of the program.
- The **second integer** specifies the variable type flag (e.g., float, int, or Vector3). This flag directs how a given placeholder should be interpreted or what type of variable should be inserted.

This templating mechanism ensures that every generated line of code adheres to a pre-defined structure, thereby enforcing consistent coding conventions. The alternating sequence of fixed literals and dynamic placeholders—along with the terminating control flags—enables both precise control over indentation and variable assignment as well as the flexibility required for generating human-readable code.

By clearly mapping the genetic representation (the 2D matrix rows) to these scaffolds, the system guarantees that each line of code is constructed with a coherent format. This directly contributes to improved readability and maintainability of the auto-generated program, which is a primary objective of this research.

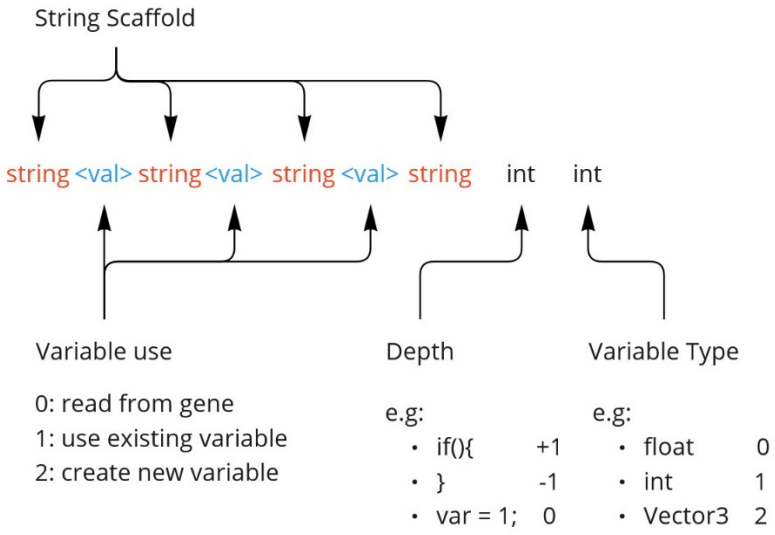


Figure 14: GLGPPS Scaffold structure

Using the genetic sequence and the library of scaffolds, we can map how the genetic matrix is used to construct an algorithm. API calls, external references, complex object parsing, multiple function calls and recursion are easily applied, when necessary, through simple additions to the scaffold library. Pseudocode for this algorithm can be found in Appendix Figure 52.

Interpreting the Genetic Sequence

Each agent's genetic sequence is stored in a variable-sized 2D matrix of integers. The matrix is structured as a list of lists, where each row (gene) contains codons of varying length. The codon length is determined by the longest codon present in the data set. The length of the gene sequence itself is also variable, as is the number of agents. This produces a list of lists of lists of integers, a potentially very large 3D data structure which is accessed by multiple simultaneous threads with both read and write operations.

The main thread batches a group from this data structure to be processed, an interpreter on this thread converts the integers of each entry in every list in every agent in its batch into source code, converting each line of the data structure into strings which follow a format of <string>, <value>, <string>, <value>… (Figure 14), where a compiled string is generated by adding sequentially each <string> object literally and filling each <value> object with either:

0: the corresponding allele value directly from the agent's genetic data for that entry

1: a named variable, selected from a list of all existing variables against a modulo of the agent's genetic data for that entry

2: a new variable is created and added the list of named variables

Each line of the genetic code (gene) will convert into a single line of source code. The first integer of each row serves as a scaffold selection index, determining the structure of that code line. The system applies a modulo operation to prevent out-of-bounds errors, ensuring selection remains within available scaffold entries. Subsequent integers in the row function as control variables, defining how placeholders in the scaffold structure are

populated, either pulling values directly from the genetic sequence or referencing existing variables.

The current model of the scaffold selection algorithm simply uses multiple instances of the same scaffold to produce the same results as assigning a likelihood to each entry, though ideally a variable occurrence frequency value would be assigned instead. This produces a form of fitness-proportionate selection [140], though on a chromosomal level rather than an agent level, allowing a heuristically lead approach which optimises the likelihood of selecting relevant chromosomes more consistently and faster when the attributed likelihoods are applied in accordance with actual fitness [141].

Different hard-coded lists of decompiled code fragments can be used for different languages, so the program can generate an arbitrary number of arbitrary length code snippets for most standard object-oriented languages.

Genetic Algorithm

To breed agents within the evolutionary process, a genetic algorithm operates to run the genetic process, handling crossover and mutation, allowing two parent agents to create a child offspring agent.

As the auto-coder operates on a 2D integer matrix, the Genetic Algorithm (GA) is also designed to operate on this data structure and utilises deep copy mechanisms to write directly into the working master genetic sequence. Otherwise, it treats each row of the core algorithm in the same way a normal linear integer GA would, so we can utilise the same optimisation techniques seen in contemporary GAs.

Crossover

The algorithm, at crossover, randomly selects between one parent's values (in this case, a list of integers) or the other parent's values. The crossover function selects gene sequences from either parent using a thread-local, static random function. This method ensures reproducibility while preventing duplication of values in a parallel execution environment. Applying crossover at the gene level retains scaffold structure, minimizing disruptive recombination, whereas allele-by-allele crossover modifies individual components, which can lead to functional degradation. This function will return 1 or 0, indicating one parent or the other, and a new genetic sequence is constructed by carrying over the gene.

This is the most simple implementation of crossover and we can expect the results to demonstrate the same regression convergence from it that we see in traditional GA's [19]. However, it can be applied on a gene-by-gene basis as well as on an allele-by-allele basis. This prior carries over entire genes, rather than single alleles as we would see in traditional linear genetic operations.

Using classical linear crossover operations on an allele-by-allele basis alone, we would expect lower survival rates in offspring and a slower convergence compared to full gene crossover, as the entire behaviour defined by the scaffold would be completely altered by its comprising alleles changing, making the crossover operation behave as a more constrained form of mutation when two dissimilar genes crossover.

Mutation

Due to the data structure being 2D, mutation can occur at either level, changing either the codon (each individual integer) or the gene (the entire row). These two operators can individually set their mutation ratios as, following the theory from Geometric Semantic Evolution [111], a full gene mutation will be more likely to have a negative impact than a codon mutation.

The mutation process itself replaces the existing value with a random new value on the codon, or a series of random new codons at the gene level.

In normal code, low integers are the most common numeric variables, reflecting human preference for these values [142], a formula was created (Figure 15) to increase the likelihood of returning low values when generating new values. This is not applied to the first entry in the genetic sequence, as that entry is used to select the scaffold the agent will use in program synthesis.

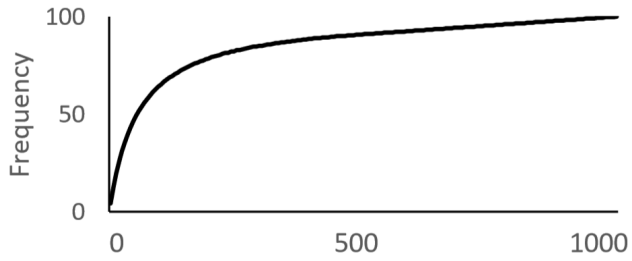


Figure 15: Cumulative frequency graph for:
$X = -7.77 + 1000 / (1 + Y / 10.15)) - 1$

Removal and insertion in invariable length genetic sequences may also be applied, if every removal applied to the GA is followed by an insertion and vice-versa, to maintain sequence length.

Removal

This process removes an entire gene from the genetic sequence, which simply removes an entry in the list at the location of the gene. This does not apply to codons separately and should be applied with a lower occurrence ratio than mutation.

Insertion

This method creates an entirely new gene, as a new list of integers. The integers are generated at random, using the same limitations as the mutation process, taking into account the first element in the sequence.

Geometric Push, Depth-Multiplicative Mutation

As the genotype compiles into a phenotype with an interpreted, non-linear structure, we can use the phenotypical structure to modify the genetic algorithm itself. This is an element inspired by Geometric Semantic Grammatical Evolution [111], which applies a

multiplier to the mutation likelihood of any one element in the genome according to the indentation depth it's phenotype would produce.

This takes the indentation depth value from the depth controller (see Depth Controller) and utilises it as a multiplier against the mutation, insertion, and removal ratios at crossover, on a line-by-line basis. The higher the indentation depth, the higher the mutation ratio. This value is scaled logarithmically to prevent severe mutation ratios at high indentation depths.

Generated Code	DEPTH	MUTATION RATIO 1/ ((Max+1) - Depth)
– if(T2 > 36) {	0	0.25
– – – if(T1 > 24) {	1	0.333
– – – – – Action1 * 85	2	0.5
– – – – – if(T2 > 14){	2	0.5
– – – – – – – if(T3 > 92){	3	1
– – – – – – – }	3	1
– – – – – }	2	0.5
– – – – – Action3 * 0	2	0.5
– – – – – Action3 * 18	2	0.5
– – – }	Auto	-
– }	Auto	-

Figure 16: Depth-based mutation ratio derived from phenotype line-by-line scope

Using the push-pop architecture, is that the indentation of the generated algorithm may be utilised as a form of geometric structure, allowing a similar approach to Geometric Semantic Grammatical Evolution [111], where the depth of the current operation may be utilised to modify the mutation ratio (Figure 16) or be utilised as an alternative form of sub-tree crossover (Figure 17).

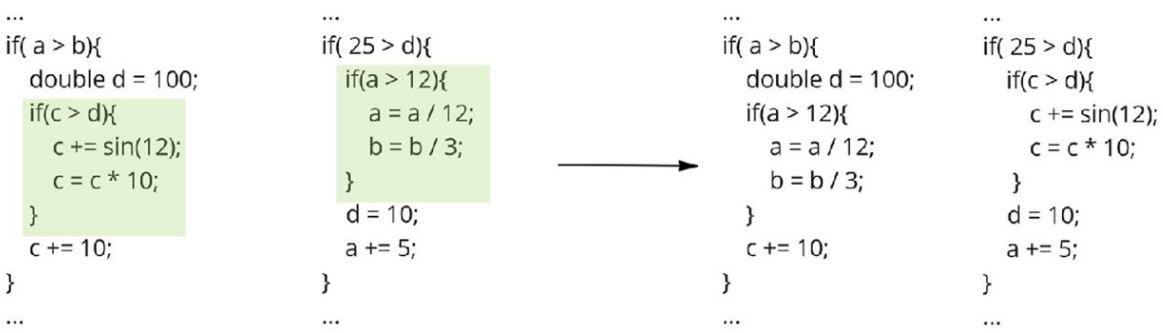


Figure 17: Example of depth-based crossover, exchanging "if" statement blocks

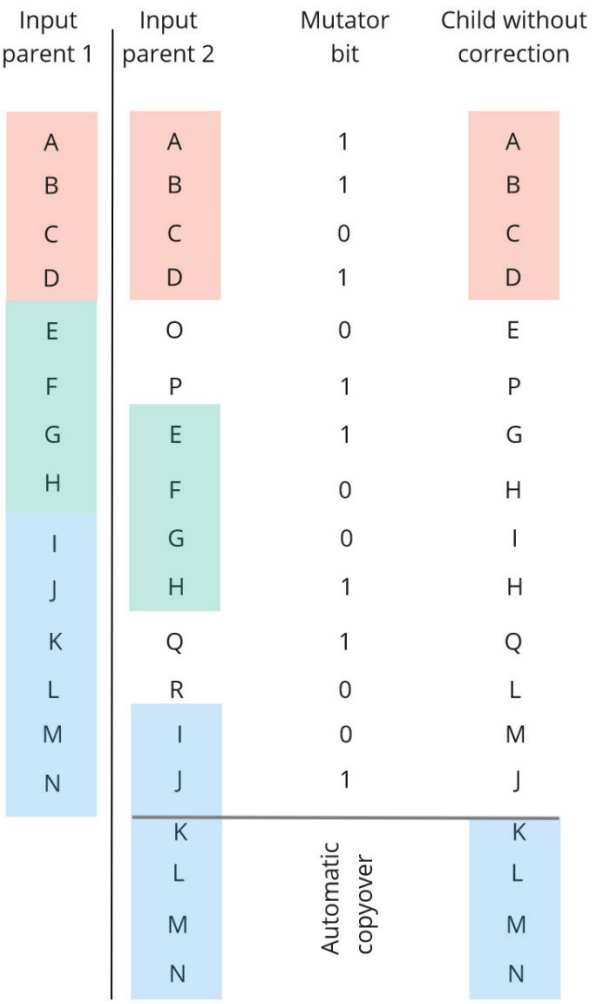


Figure 18: Two agents, Parent 1 without insertion, parent 2 with insertion, generating a structurally damaged offspring.

Correction (2D Needleman-Wunsch)

Variable-length genetic sequences introduce fragmentation challenges during crossover, where insertion and removal operations cause gene misalignment. Without correction, even nearly identical parents can produce offspring with disrupted structure. We explored several methods—truncation, code transfer, and random sampling—to realign these sequences. While truncation preserves structure, it risks prematurely shrinking the search space; random sampling, though diverse, often yields fragmented outputs. Our preferred strategy dynamically selects one parent's gene length to standardize sequence size, thus maintaining genetic stability.

Although a modified Needleman-Wunsch realignment algorithm was implemented as a 2D correction mechanism, consistently preserving both gene-level and allele-level integrity during crossover, leading to improved speciation rates and enabling the generation of functionally coherent offspring (Figure 18).

Addressing crossover misalignment, several correction methods were explored: truncation, code transfer, and random sampling. Truncation risks shrinking search spaces prematurely, while random sampling disrupts structural integrity. The preferred approach dynamically selects one parent's sequence length to maintain genetic stability.

Handling agents who are not of equal size thus becomes a design consideration due to this correction process. Options would comprise either truncating all agents to the length of the shortest agent, copying over the code from the longest agent or randomly sampling from the longest agent. Truncation alone is likely to lead to rapidly diminishing genetic sequence length, which may encourage limited solution search spaces. Random sampling from the longest agent is likely to cause more loss of structure, so the preferable result is to randomly select between the length of one of the parents.

Figure 19: Algorithm for identifying all permutations of two aligned sequences

```
1.  Function GenerateArrangements(content, n, arrangement, currentIndex, output):
2.      If (size(arrangement) == n) Then
3.          contentSet ← CreateSet(content)
4.          arrangementSet ← CreateSet(arrangement)
5.          If (contentSet ⊆ arrangementSet) then
6.              output ← output + Join(",", arrangement) + "."
7.          End If
8.          Return
9.      end If
10.     Append(arrangement, -1) // Add a blank entry (space)
11.     GenerateArrangements(content, n, arrangement, currentIndex, output)
12.     RemoveLast(arrangement)
13.     For (i ← currentIndex) to (size(content)) Do
14.         Append(arrangement, content[i])
15.         GenerateArrangements(content, n, arrangement, i + 1, output)
16.         RemoveLast(arrangement)
17.     end For
18. end function
```

A Needleman-Wunsch re-alignment algorithm was produced (Appendix, Figure 51) with limited application. In most experiments, insertion and removal were applied at the same time to create agents with consistent size, so one gene insertion would always be followed by one gene removal, to mitigate the impact of the automatic copy-over effect.

This 2D representation sees most of the manipulation at a gene-by-gene level as the most significant change to an agent's genotype arrives from gene level crossover algorithms. A modified implementation of the Needleman-Wunsch algorithm was created as a correction method

As the correction that this algorithm looks to resolve is gene level as well as allele level, a new support algorithm was created to generate every permutation of alignment of genes which maintain the order of both sequences created and do not remove or add any genes.

This recursive algorithm (Figure 19) generates a series of pairs of integers which identify all permutations of the two aligned sequences with worst-case complexity of $O(2\hat{}m)$. This complexity makes his algorithm unsuitable for larger searches with constrained compute or time allocation and the resulting search for smaller samples did not demonstrate substantial

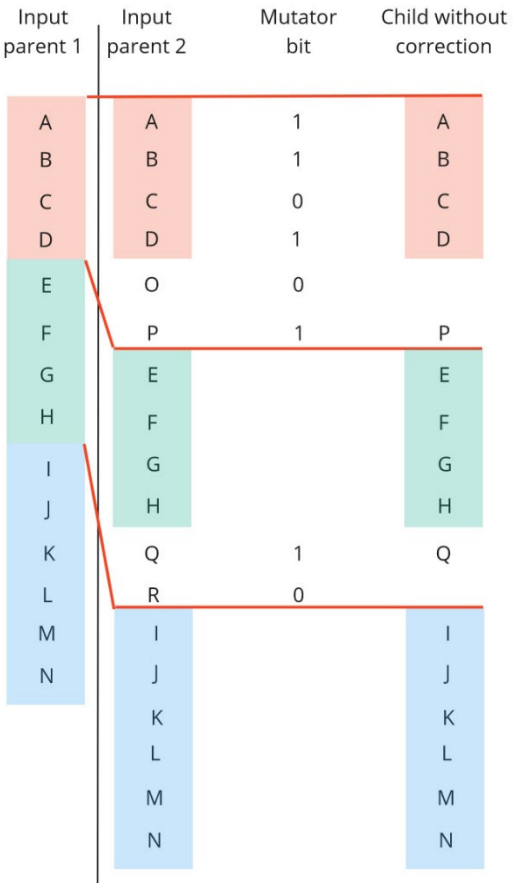


Figure 20: Alignment Correction derived from similarity in scaffold ID values.

Using this list of pairs of arrangements, a novel gene (rather than allele) based alignment variation of Needleman-Wunsch is used for crossover. This algorithm checks if the tested gene is identical, different at all or does not match to an existing pair for best alignment. This algorithm selects randomly between which element between both genes to add to the output string during trace-back.

A simple visualisation of this method in action can be seen in Figure 20. This represents, at a high level, how this can be used on a scaffold-by-scaffold basis. This would identify similar code blocks using the first allele in each codon and grouping codons of high enough similarity, permitting direct carryover between them. This would allow two genomes of different length to crossover while maintaining functionality, without a crossover mechanism to correct misalignments when injection and removal is present, the child would more often produce noisy solutions, losing the core functionality of the whole solution, due to a single variation in gene length.

After the first value (which is used to select the scaffold), entries of each line of an agent's genetic sequence are used to fill the variable information in the scaffold. How these variables are used depend on an integer within the scaffold itself. The scaffold may have multiple, different corresponding variable control integers, currently these integers represent the following use-cases:

0 (Direct Insertion): The genetic value is used directly and literally; it is inserted as a string directly into the output source code.

1 (Variable Reference): The genetic value is used against a list of existing variables. Modulo is used to prevent out of bounds reference exceptions. The available list of existing variables is dependent on a scope controller and variable ID mechanism, so the same codon will not necessarily produce the same phenotype, as it is dependent on the preceding code.

2 (New Variable Creation): A Request is made to create a new variable. The genetic sequence is not applied here, instead, a process is applied which creates a new variable within the source code, which proceeding calls to use an existing variable will have access.

For both variable referencing and creation, a dedicated value formatter converts integers into consistent strings. Pre-defined variables—such as input parameters (e.g., *hunger*, *age*, distance metrics in simulations, or constants like π, e, or g)—occupy the initial positions in the available variable list and are treated identically to automatically defined variables once established.

Figure 21 demonstrates an example gene with alleles *8,7,4* mapping against the scaffold library. Scaffold *8* is selected, corresponding to the first allele in the gene. This scaffold holds a string representation of the main body of a line of code and, in this case, has two variable control integers, 2 and 0 (create new variable and use gene value directly). As a new variable was requested, the current counter of existing variables is incremented and a new variable is added to accessible variables the list, using a simple string generator with a list of banned variable names.

The output is a complete line of code (*Double G = 4;*). Although concatenating these lines produces source code, this method lacks scope control and context-sensitive structural delineation. Without proper scope management, conditionals and loops cannot be safely integrated, as their correct interpretation depends on hierarchical structure.

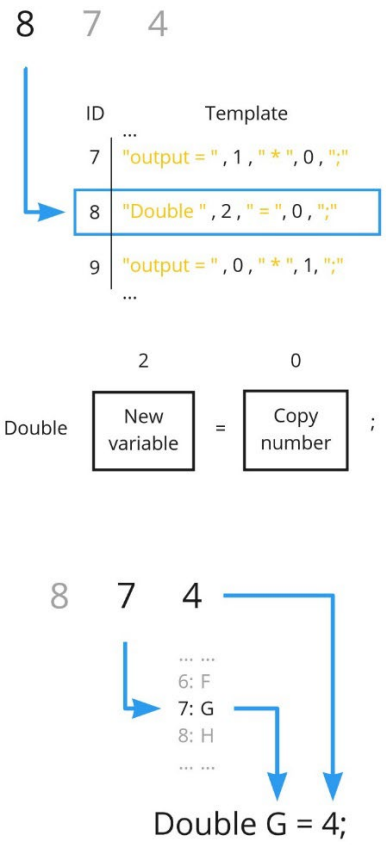


Figure 21: Process of interpreting the genetic sequence "8,7,4", without depth control, to source code.

Depth Controller

To achieve robust scope control, GLGPPS incorporates a push-pop stack system (Figure 22). This allows both proper indentation for source code legibility and proper variable scope control, so it can prevent variables instantiated inside a block from being accessed outside of that block.

This depth handling algorithm (Figure 23) also allows us to expose an element utilised in Geometric Semantic Evolution [111], directly to source code. We can use the depth of indentation to assume diminishing return of damaging mutations to linearly beneficial mutations at increased depth. We can modify the mutation ratio proportionally to the indentation depth, to optimise the gradient between beneficial mutations and detrimental mutations.

This "geometric push" mechanism is distinct from Lee Spector's Push algorithm [130], though this algorithm does use multiple push-pop stacks for multiple variable type control. The scope controller maintains a dedicated stack by pushing variables as they are introduced and popping them once their defining scope ends. A variable can only be referenced if it remains in the stack, with its reference determined by its positional index in the genetic sequence. This approach enables dynamic management of in-scope variables while preventing null references and type conflicts.

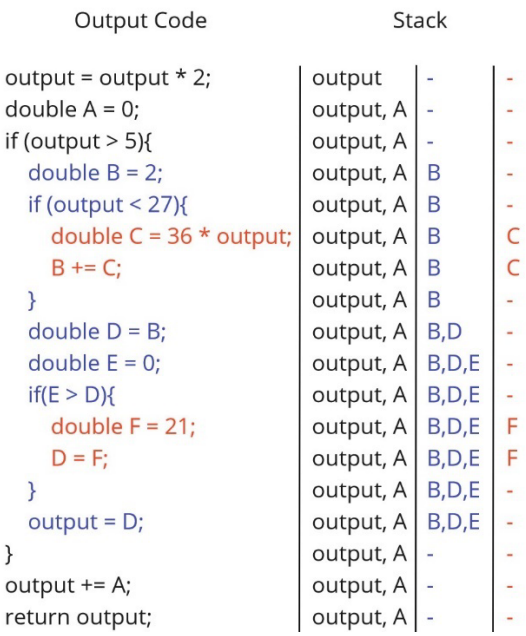


Figure 22: Example of scope as seen by depth controller stack.

This stack works using a push-pop stack architecture: a push occurs when the scope is incremented (for example, an opening bracer from an 'if' statement), the push increments the depth and allocates all following code into the current stack, until another push or a pop occurs. A pop occurs when the scope is decremented (for example a lone closing bracer '}'), which removes the highest-level stack from accessible scope. If a new push occurs, the previous code at the same level stack is ignored.

A depth counter is used to determine the current depth of the scope, if the genetic sequence terminates and the stack depth is greater than 0 (the default minimum stack depth), a cascade of closing bracers is applied for every unresolved depth. This may be removed while retaining scope control for languages who do not require explicit termination of scope.

The number of tabs used to produce indentation for visual consistency also defines the stack depth. At the beginning of each line of code, an indentation is produced proportionally to the current stack depth, producing visual indentation.

Another benefit of this system is the use of the 'else' statement. If the depth is greater than 0, and the only existing depth incrementors are compatible with 'else' statements (e.g., a complimentary 'if' statement), we have an environment where an 'else' statement is syntactically valid. 'Else' statements do not modify depth but are handled by being placed one indentation lower, as they both end and begin an indentation, so the following line of code will retain the same indentation depth as the previous line.

Notably, a modification will need to be made to the stack architecture (a simple binary check) for each depth to define if its 'push' was associated with an associated starting structure. For example, a 'for' loop is not compatible with an 'else' statement. This would be easy to implement, though this model does not use loops to reduce the likelihood of a subspecies who utilises a recursive, non-functional, multi-layer 'for' loop which uses a

disproportionately heavy CPU cycle. This method should be included in auto-coders whose solution space benefits for alternative indentation patterns.

Looping over these systems for every line of the genetic sequence, we can produce a fully functional, compilable string of source code.

Figure 23: Algorithm for controlling code indentation

```
19.     if ((indentDepth + 1) > Size(stack)) then
20.         Append(stack, 0)
21.     else if ((indentDepth + 1) < Size(stack)) then
22.         while ((indentDepth + 1) < Size(stack)) do
23.             stackTopValue ← stack[LastIndex(stack)]
24.             valuesAnnounced ← valuesAnnounced - stackTopValue
25.             for (i ← 0 to (stackTopValue - 1)) do
26.                 RemoveLast(valueList)
27.             end for
28.             RemoveLast(stack)
29.         end while
30.     else if (valuesLastTick < valuesAnnounced) then
31.         stack[LastIndex(stack)] ← stack[LastIndex(stack)] + 1
32.     end if
33.     valuesLastTick ← valuesAnnounced
```

Multiple variable types

As the agents for the AL implementation will be dealing with co-ordinate geometry, while also operating primarily with floating-point values, a method is introduced to handle multiple data types as independent object structures. Currently, these are handled at the scaffolding level (*Figure 24*), as the data type and how it is sourced is defined as an integer within the scaffold. For this simulation, the rightmost integer and associated operator are used to control indentation, with + adding and – subtracting.

The following variable control integers are used for variable insertion:

0 = Use value directly from corresponding genetic sequence
1 = Use existing float from existing floats table
2 = Create and assign new float to floats table
3 = Use existing Vector3 from Vector3 list

Figure 24: Examples of code scaffolds

```
1.  new List<string> { "1", " = ArrayDistance(", "3",",","3",");","0"},
2.  new List<string> { "target= ","3",";", "0"},
3.  new List<string> { "target = Lerp(", "3",", ", "3", ", -1);", "0"},
4.  new List<string> { "3"," = ","3",";", "0"},
5.  new List<string> { "3"," = Lerp(", "3",", ", "3", ", -1);", "0"},
6.  new List<string> { "if (", "1"," > ", "1", "){", "+1"},
7.  new List<string> { "if (", "1"," < ", "1", "){", "+1"},
8.  new List<string> { "if (", "1"," > ", "0", "){", "+1"},
9.  new List<string> { "if(", "1", " < ", "0", "){", "+1",
10. new List<string> { "if (breedCount Down <", "0", "){", "+1"},
11. new List<string> { "if (breedCount Down <", "1", "){", "+1"},
12. new List<string> { "if (ArrayDistance (", "3",", ", "3",") > ", "0","){","+1"},
```

The algorithm is not limited to utilising these four variable control integers, it is possible to add more, by simply adding a reference value and directly coding how that new data type should be modified in the interpreter itself.

It should be noted that this allows Vector3's to be modified directly but does not currently allow new Vector3's to be generated. While it is possible to do this, it would require modifying the scope controller, using a 2D matrix which only allows access to variables if they are both in scope and of the associated data type, so that generated variables may be called in the appropriate sequential order without counting inaccessible data types.

For loops are created with a counter set outside the testing function, which increments for every iteration of any loop inside the tested function. This prevents infinite or very high recursion scenarios. When testing, agents who time out are given a zero score.

Genetic Summary

The Gene-Level Geometric-Push Program-Synthesis (GLGPPS) algorithm represents source code as a two-dimensional genetic sequence, where each row corresponds to a complete line of code. Within this framework, evolution is driven genetic operators, including crossover, mutation, gene insertion, and gene removal, which explore the solution space. In the GLGPPS model, the crossover mechanism operates at the gene level, enabling the exchange of entire code lines between parent agents. This gene-level recombination maintains the integrity of scaffold-based templates—ensuring that structural and syntactic coherence is preserved throughout the evolutionary process.

Figure 25 illustrates the application of these operators in practice, demonstrating how gene-level crossover, coupled with mutation and gene removal, produces a new child agent from a pair of parents. The crossover operation selects complete genes to be swapped, which guarantees that the fundamental building blocks of the generated source code remain intact. In parallel, mutation is applied at both the allele and gene levels, providing fine-grained and structural modifications respectively, while removal operators adjusted the overall length of some genes.

```
Parent A:

[1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1]
[1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1]
[1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1]
[1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1]
[1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1]
[1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1]
Parent B:

[2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2]
[2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2]
[2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2]
[2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2]
[2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2]
[2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2]
CHILD:

[1, 13, 1, 1, 1][1, 1, 1, 1, 1][2, 2, 2, 2, 2][1, 9, 1, 1, 1][2, 2, 2, 2, 2]
[1, 1, 17, 1, 1][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2]
[2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][1, 1, 1, 1, 1]
[2, 2, 2, 2, 2][2, 2, 2, 2, 2][2, 2, 2, 2, 2][1, 1, 1, 1, 1][2, 2, 2, 2, 2][50, 1, 1, 1, 1]
[1, 1, 1, 1, 1][1, 1, 1, 1, 1][2, 2, 2, 2, 2][1, 1, 1, 1, 1][1, 1, 1, 1, 1]
[2, 2, 2, 2, 2][2, 2, 2, 2, 2][1, 1, 1, 1, 1][1, 1, 1, 1, 1][1, 1, 1, 1, 1][2, 2, 2, 2, 2]
```

Figure 25: Complete crossover mechanism with mutation, insertion, and removal across multiple genes

Templating Mechanism

Addressing a need for rapid dynamic evolution with likely anticipated behavioural factors in a virtual ecosystem, a Template-Based variant (TB-GLGPPS), taking inspiration from Template Based Evolution (TBE) [124] was developed. A subsumption architecture can be generated within the source code. This architecture can be enacted as multiple function calls from a main function which subsumes a series of functions.

The 2D data structure defines a function as each row of integers (alleles) compiles into a line of code (genes), and a series of lines of codes creates a function (chromosomes). To permit the execution of multiple functionalities from within the source code, multiple functions are used, creating a 3D data structure (Figure 26).

The templating architecture works with a multi-layer Augmented Finite State Machine (AFSM) (Figure 26). The values of the AFSM are read from the genetic code of the agent, using the first line of the agent's genetic sequence (a, b, c) as its trigger threshold. Each of these thresholds are testing against a sensor input or a function output. The AFSM enacts function calls against the auto-coder's automatically generated functions (A, B, C, D, E).

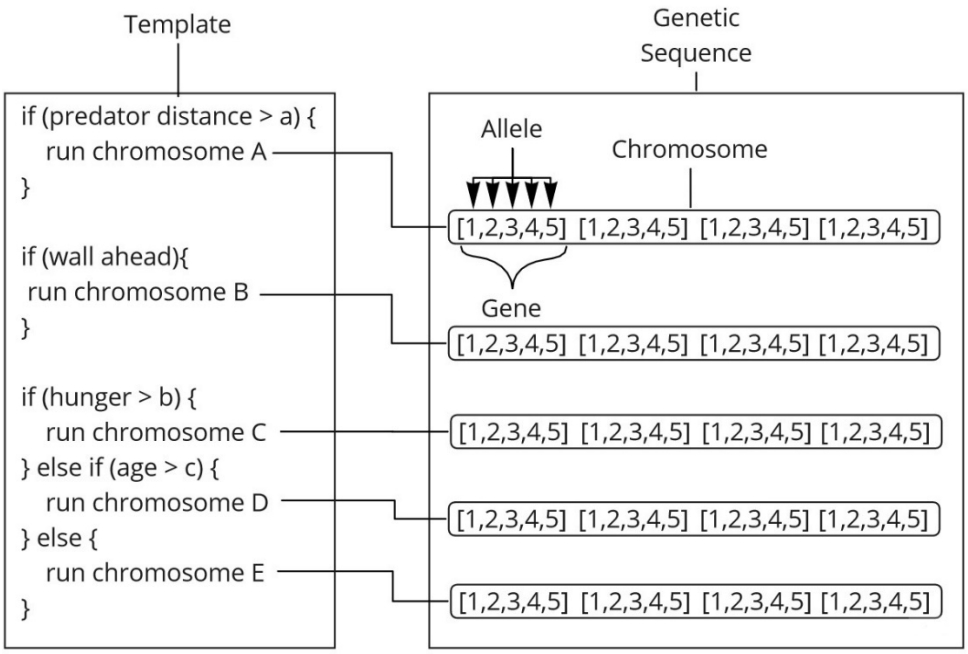


Figure 26: Attaching templating mechanism (left) to a 3D Genetic Sequence (right)

We can also apply the output of one of the auto-code functions against another as part of the AFSM. This simply compares the output of the two behaviours as a determinant for the trigger of another auto-code script.

This mechanism will implicitly bind the behaviours of the agents to work around hard-coded sensor input constraints, though still leaves the behaviours adopted to each of these constraints open. This templating architecture assumes existing knowledge about the anticipated structure of the output source code as the AFSM's structure and the input comparatives are hard coded, so is only applied when explicit intervention to enforce assumed pre-requisites of the algorithm should be applied, or when a general structure is known or preferable – a hard-coded template is used for this thesis.

Analysing the prospective search space, this will enforce a higher dimensionality by default, so when designing the templating system for implicit fitness functions, those functions must either be substantial to an agent's value, or their use should be limited. If too many are applied without significant value, the agents will be likely to experience loss of function mutation, though this will generally propagate in code which is less critical to a species survival. Notably, it is likely that as mutation rate increases, the likelihood of loss of function mutation or the formation of amplified code will increase proportionally.

Automatic unused-variable removal

GLGPPS is subject to producing unused variables or redundant behaviour, "bloat", assignments and statements which do not in any way modify the output of a generated function. This generated code bloats the output, harms legibility and comprehension.

Classically, static code analysis tools, often using a linter, will identify unused variables from an abstract syntax tree representation of the code.

A simple variable removal algorithm is implemented for GLGPPS which identifies unused variables, by tracing backwards from the *output* variable, line by line. Active variables are held in a stack, which are removed when announced. This implementation is novel in evolutionary program synthesis.

As a line is analysed, if the variable des not contribute towards the output variables or any contributing variable met so far, the variable is pruned.

An exception is made for terminating bracers, which search the pairing statement announcement for any control values, which are put into a separate stack. Any value which modifies any variable which is not created inside the loop but is modified inside the loop is not pruned. If a statement block does not modify any external variables, the block is removed. A specific statement is also made for “for” loops, where the search limit is removed.

This implementation is simple but operational in the context of the problem spaces addressed and the scaffolds used for the experiments in this thesis. A more complete code analysis tool is recommended in future iterations of this algorithm for improved output comprehension – it is likely an existing linter could be integrated directly in future iterations.

4.4: Server Architecture

To support a co-evolutionary artificial life and human-centric program synthesis platform with multi-user connectivity, a robust, real-time simulation framework was implemented. The server architecture, bult for the GLGPPS system, provides centralized control over the evolution of agents, environmental simulation, and real-time client interactions.

For multiple simultaneous users to collaborate within the same virtual environment, the virtual world experienced by all simultaneous end users is received, as data, from a server (Figure 27). This centralised server handles the sending and receiving of data from users, and the entire Artificial Life simulation: the evolution of agents, the model of the terrain and the behaviour and physics of agents.

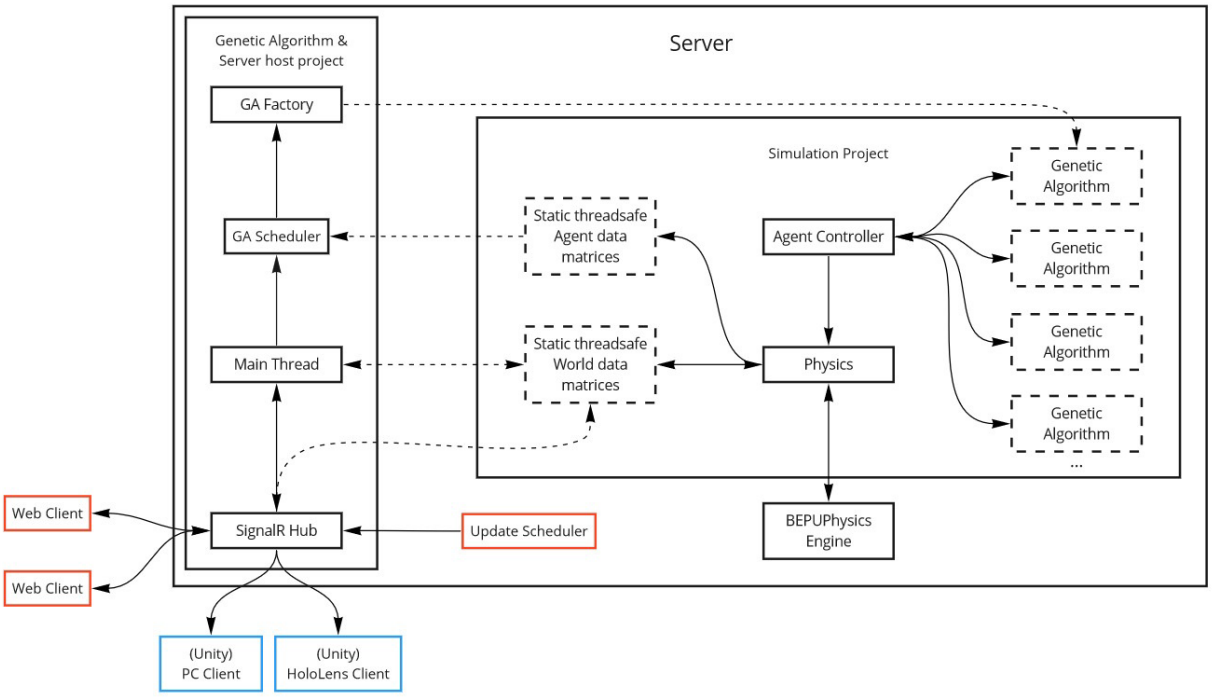


Figure 27: Server architecture diagram

The core application was an extension of the central GLGPPS server, the system ensures consistent performance, scalable real-time operation, and synchronized user experiences in a collaborative virtual ecosystem.

A Unity application is created for client-side rendering and interaction, where an application can easily be exported to the HoloLens for the Mixed Reality interface for rendering. This model reproduces, with enhanced visual effects, the geometry of the terrain and the location of agents, from packages received from the server. It also provides a platform to transmit data back to the server, allowing requests for human modification of the virtual environment on the server, from the client.

The application's server runs on dual Xeon E5-2670 v2 processors, with 20 CPU cores before multithreading. This server allows space for a highly parallelised software design philosophy around the hardware, however, does not compare to modern server frameworks for benchmarking or generation of large program synthesis models.

To handle the world simulation on this system, a modern, efficient SIMD – accelerated [143], CPU-oriented, .Net Core [144] physics engine: BEPUphysics [145], was incorporated, though it was also in prototype phase and was a relatively undocumented, UI-less, code-only engine, it came with many novel physics optimisations for highly parallel servers and was capable of integrating directly with the existing server program. This physics engine project was inherited by the main project, for direct manipulation within a single application.

Convert-Compile-Commit-Delegate

To execute this source code within the same application as it was generated, at run time, we use a method that allows dynamic compilation and execution. As this algorithm is executed in C# (for improved runtime speed with repeated execution of compiled code), an intermediate language, code cannot be directly executed as a string, as it could in a fully interpreted language.

Due to C# being non-interpreted, to dynamically generate and execute source code, live, required use of the Roslyn dynamic compiler [146], to create virtual assemblies or Dynamic Link Libraries (DLLs), a file containing an accessible library of functions.

To generate these DLLs, we first convert the source code into a C# compilation, we begin by generating a syntax tree using the code analysis library [147] from source code. This is then compiled and committed into a memory stream as a RAM only DLL.

The algorithm was developed in .Net Core 3.0 preview 1, which had just released at the point of development of this thesis. This environment was chosen for performance[148], [149] server architecture designed to handle many simultaneous high bandwidth calls on a highly multithreaded architecture and it's intermediate language parsing. This pushed the development of the program into C#, a language which is not interpreted, making direct code injection impossible, though this is addressed with a convert-compile-emit-delegate process.

Fortunately, the version of .Net Core had just introduced a prototype of this compiler, though early tests with this model, mass producing DLLs directly into RAM, rapidly filled system memory, without a process for clearing memory. Fortunately, this version of .Net core also just introduced collectable assemblies [150], allowing the rapid destruction of unused DLLs.

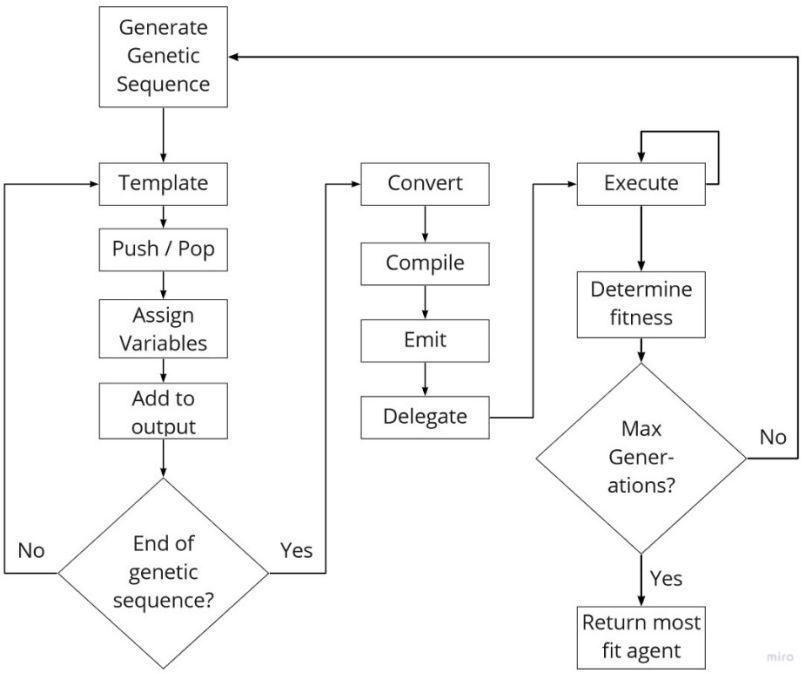


Figure 28: Dynamic compilation, multiple generation process overview

To access these automatically generated, in-memory DLLs and call them similarly to a normal function, C#'s reflection API [151] is used. The memory stream is read directly into a context interpreter – a reflection library that is used here to expose the functions within the generated DLL. A delegate is generated out of the function call, which is then stored as a collectible delegate in an array. When referenced as a delegate [152], the performance of these functions are nearly identical, at run-time, to the original code of the application. This algorithm is multithreaded with performance linear to core count.

To accommodate multiple functionalities, several functions are implemented within a single file, resulting in a three-dimensional data structure—specifically, an array of two-dimensional arrays. This design necessitates modifications in the execution of the genetic algorithm, as the variable length of each function requires that a separate genetic algorithm be applied for crossover on each function.

The outcome of this algorithm (Figure 28) is an entirely new algorithm for genetic programming, a 2D Geometric-Push evolution for source code, capable of: 'if', 'else', 'for', 'while', recursion, access to all APIs, automatic creation and registration of variables and control over multiple data types, with relative language agnosticism and a method for generating and executing code within a single application.

Web Client Architecture

The main server application runs on a .Net Core 3.0 MVC [153] server architecture (Figure 29). This architecture was chosen as it is designed for website and web-applications and can handle very large data throughput and processing for scalable hardware implementations, allowing easy communications to multiple sources.

The application uses the MVC architecture to feed a web page, which is used to allow access to the server from multiple simultaneous external sources. This web page is a simple HTML page with a text box input and a text area output which automatically updates its content against the server's output, allowing commands to be executed from the web host and feedback to be returned. This allowed a live update of the state of execution of the core AI and allowed the generated source code files to be dumped as text to the web client.

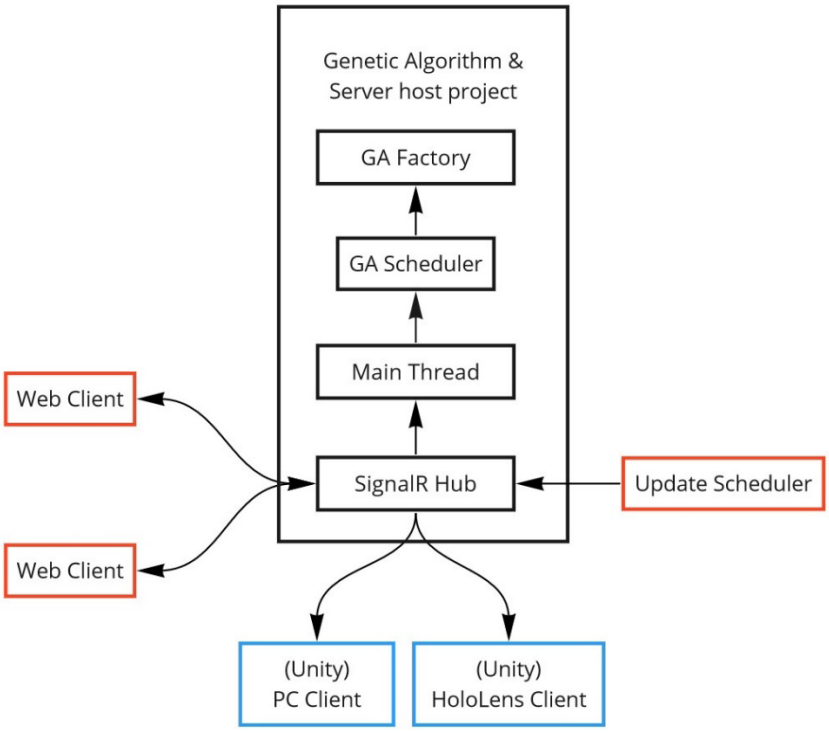


Figure 29: Agent and Server Host Architecture

SignalR Hub

The server communicates to external clients as both a generic web page spinner and Server-Sent-Events [154] host using ASP.NET Core SignalR [155], allowing unidirectional data transmission without polling from the server, without an explicit bi-directional port input for the web page. This also allows input to be handled by a controller in the MVC server, allowing data to be more safely parsed. The same infrastructure is used for the Unity-based client, which has an implementation of a Core SignalR transceiver [156], allowing the two different output mediums to share the same SignalR hub, reducing the processing overhead on the server to a single, common implementation.

The web host can execute a series of commands:

- Start physics and agent simulation,
- Manual termination of current simulation,
- Construct new, empty generation of agents with randomised initial constraints,
- Construct, destroy, get, or set agent at position X,
- Breed agent X with agent Y,
- Run evolutionary process with X number of agents for Y generations to resolve equation Z,
- Run with crossover settings:

Depth based variance– Utilising a similar technique to Geometric Semantic Grammatical Evolution [111], we can vary the likelihood of mutation in accordance with

the indentation depth of the resultant source code. The current depth that a gene operates in the resultant source code may be proportional to the mutation likelihood.

Removal – a ratio for the likelihood per generation of removing a line of code at a geometrically dependant random location within the genome.

Insertion - a ratio for the likelihood per generation of inserting a new line of code at a geometrically dependant random location within the genome.

Normalised random – to simulate the outcome of more complex subsets of genes at crossover, the resultant output of a crossover event may be partially normalised or randomly deviated, allowing the result to either be a direct inheritance or, with a set likelihood, given a slight deviancy from the original value.

Manual garbage collection – due to the core AI's reliance on high volume, high churn rate RAM usage, with large numbers of relatively large data objects as loosely coupled DLLs, a manual request for batched finalizers and garbage collection may become necessary in very large-scale execution runs. This is applicable where automatic garbage collections fail to reclaim at a high enough rate to match RAM usage alone. This only occurs in large parallel execute implementations, where many agents are run for many generations.

Main Thread & Schedulers

Upon requesting execution of either the evolutionary process or the main simulation, a new thread is instantiated which controls the execution process. This thread lasts for the entire duration of the execution of the requested process and handles scheduling, client requests, thread spooling and data handling.

When executing the core AI independently, the thread divides the number of agents into the number of virtual cores, allocating a new thread for each virtual core to evenly distribute workload, generating a new C# file for each of their allocated agents according to the GLGPPS construction framework. These files are then executed following the convert, compile, emit, delegate sequence and benchmarked.

The main thread then executes a deep copy of the data of each thread's genetic sequences, due to the main matrix being a complex data object. This is done by directly serialising a memory stream over the original data, into the main genetic data matrix, directly overwriting the previous generations genetic data with the latest generations. It then assigns the breeding order according to the result of the benchmarks and assigns the next generation of agent controlling threads. After a pre-set number of generations or on the occurrence of a termination event, the main thread returns the agent with the highest scoring benchmark.

When executing the simulation, this thread starts a slow schedular, a long running polling thread between the web server project and the child simulation project, allowing breed requests to be identified in the simulation. The polling thread then instantiates a series of threads and divides the requests against the number of virtual cores in the CPU, to breed the genetic sequences of these agents together, using the same processes as executing the AI independently, though without any benchmarking and only for the agents who had been requested explicitly to be generated. This thread then sets the matrix in the

simulation project, holding the delegates pointing to the generated DLL's and directs them to each agent's new DLL. This thread and its polling thread run until manually terminated.

Algorithm Overview

The GLGPPS algorithm represents source code as a structured sequence of genetic information, where each gene corresponds to a single line of code. A scaffold system translates integer-encoded genes into valid source code by using the first allele to select a predefined template and subsequent integers to populate variable fields. This modular representation decomposes a program into discrete elements that maintain syntactic and semantic integrity.

The core genetic algorithm applies gene-level operators including crossover, mutation, insertion, and removal. Crossover exchanges entire genes between parent agents to sustain structural consistency, while mutation is performed at both the allele and gene levels. The push mechanism and correction routines based on sequence alignment on variable-length genetic sequences, ensure compilation and coherence.

The process is integrated into a centralized server architecture implemented in C#.NET Core. This server manages the evolution of agents, simulation of terrain and physics, and real-time client communication. By centrally processing genetic operations and simulation data, the system supports automated source-code generation, execution, and subsequent modifications in a collaborative virtual environment.

This defines an approach not yet seen by the GP community; however, while this appears like a generic architecture for software problems, this algorithm still fails to completely address Michael O'Neill and Lee Spector's 'Open Issue'[126] or Yampolskiy's principal argument against genetic program inference [125] – it's trapped within the confines of regression and has exponential processing requirements for linear increases in problem complexity. Utilisation of hard coded or automatically inferred code snippets and alternative evolutionary mechanisms and method for automatically defining the appropriate layers to mitigate this partially are explored in Future Work.

Chapter 5. Evaluation

This chapter evaluates the Gene-Level Geometric-Push Program-Synthesis (GLGPPS) algorithm, assessing performance in comparison to established benchmarks, identifying strengths and limitations, and demonstrating applicability in both traditional program synthesis tasks and creative domains. The chapter begins by outlining the experimental setup and the scope of evaluation. It then reports results across a range of benchmarks: classical search problems such as OneMax and Single-Point Hill Climbing, the Helmuth and Spector program synthesis suite, artificial life simulations, and a live co-evolutionary SuperCollider music generation system. The chapter concludes with a discussion of the results and the limitations of this evaluation.

Limitations: The versions of GLGPPS tested in the following benchmarks do not utilise classical optimisation strategies or alternative ranking algorithms. The following experiments use a simple elitest ranking. However, GLGPPS is compatible with a range of optimisation and search space optimisation strategies for classical genetic algorithms, including prolific strategies in comparative benchmarks such as tournament selection.

This version of GLGPPS demonstrates a novel method for program synthesis. It does not aim to surpass compilation speed, accuracy or completion over existing methods, nor does it attempt to resolve arbitrary program synthesis.

5.1: Experimental Setup

Benchmarking GLGPPS as a program synthesis tool, a series of experiments are presented from existing benchmarking standards across the field:

OneMax [73] tests the performance of the algorithm at searching an allele-level fitness landscape. This algorithm searches for the largest possible sum of a bitstring: to evolve a series of values from a random initial state into a series which consists entirely of ones. A brief assessment is also performed on the Needleman-Wunsch correction algorithm for OneMax.

Single-point Hill Climbing, a single, consistent target value is given across all tests, no inputs are given. This test is simple but a classical exploration for genetic systems to demonstrate speed and accuracy of convergence.

Helmuth and Spector's benchmark suite [7], [157] for program synthesis algorithms, derived from an introductory computer science textbook [158] and a series of benchmarking challenge used originally for automatic program repair [159], the current standard for performance benchmarking in this field. This suite provides a collection of algorithmic problems as a standardised series of input values given as inputs to the program synthesis algorithm and a series of expected output values. A program is scored on the number of successful completions from a large testing set after completing a restricted training set. This paper stresses that program synthesis algorithms are widely

variable in operation and method, using program success as a metric, though also supplies parameters. This benchmark is used only for the input and output terminals operational in this version of GLGPPS.

Artificial Life explores the algorithm in an artificial life simulation, to explore generated code complexity in an explicit fitness scenario given variable parameter.

SuperCollider Music Generation (Autopia) analyses GLGPPS as a music generator, in a language independent of the host platform, working with humans as part of a co-evolutionary music generation artefact.

Experiment 1: OneMax

This experiment uses Bremermann's OneMax [73] algorithm in which each allele match between a candidate and the target string increments the score. Our goal is not to optimize OneMax itself, but to use it as an abstract testbed for GLGPPS to test the evolutionary functions ability to converge.

As GLGPPS operates at both the allele and the genome level independently, functionally working on the search space in two dimensions, it justifies testing convergence speed of the crossover method as a baseline. This implementation does not look to optimise OneMax but explore the quantitative relationship of the fixed length allele per variable length gene search for a 2D (Gene-Level) representation of a genetic sequence due to the unconventional representation of GLGPPS.

This algorithm assesses the genotype of the agents, but no compiled phenotype is assessed in this experiment. This algorithm is therefore directly analysing the crossover for allele-level complete alignment. As this algorithm constructs from a library with N values, rather than binary, we also look at an example with 10 possible values for the integer, again looking to identify a solution of all-ones, to represent a realistic heuristic search space.

Parameter	Setting
Search	**OneMax**
Run count	3
Agent count	800
Generation Maximum	100
Bitstring Length	$n = (20,30,40 \ldots,150)$
Mutation Ratio	2%
Injection Ratio	1%
Variable modifier	Mutation 3%, Injection 3%
Starting length	= target length

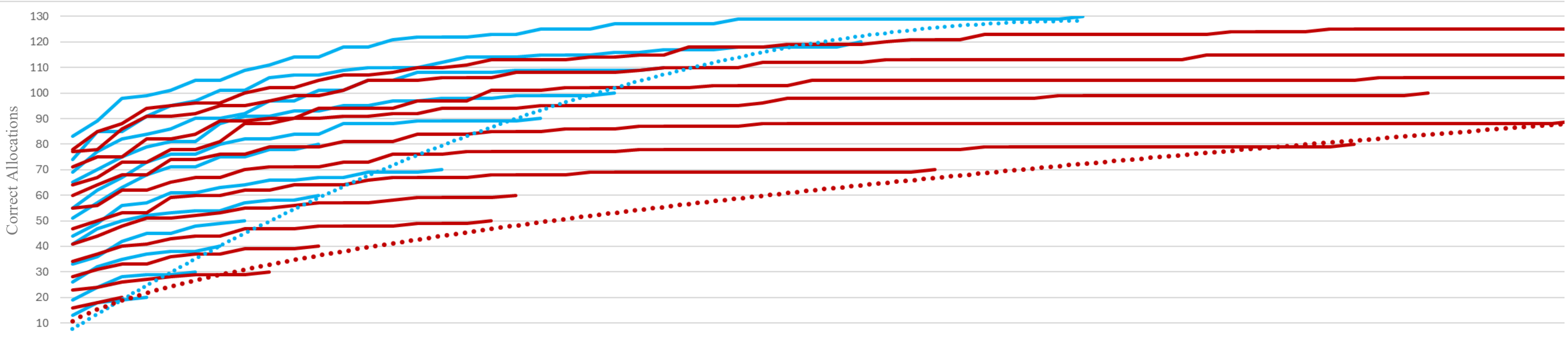
Table 1:genetic algorithm parameters for ONEMAX Benchmark experiment

Figure 30 graphs the results for a progressive series of searches in increments of 10 values to solve. This demonstrates a search for both binary and decimal search spaces for these increments, demonstrating compute change for variable lengths of the scaffolding library.

Whan analysing the results for the number of generations taken to solve this set number of binary allocations, an exponential relationship is generated for both scenarios:

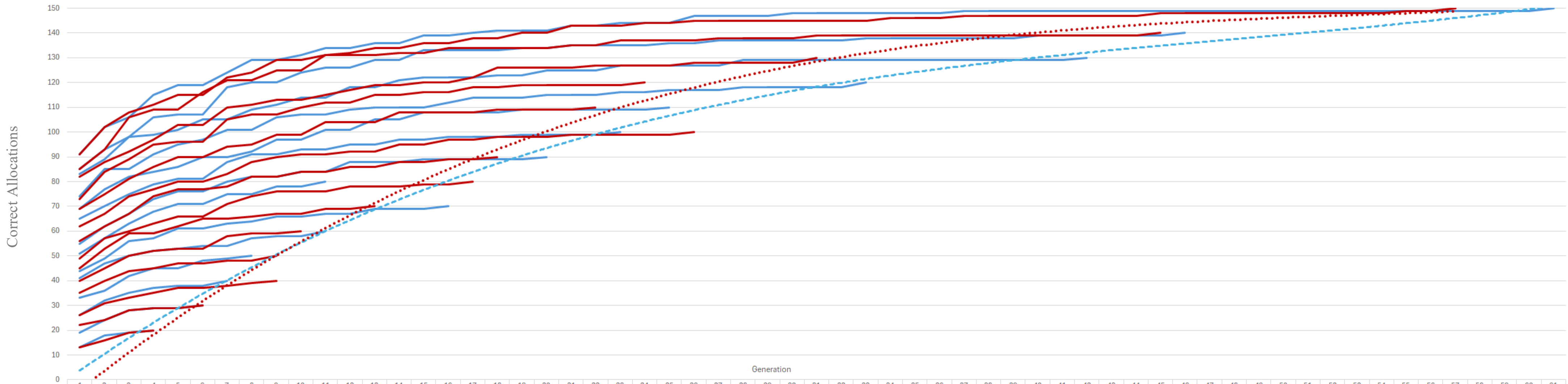
	Binary	10-scaffold library
Curve:	$y = 3 * 1.02^x$	$y = 3 * 1.0\dot{3}^x$
R^2 value:	0.96	0.97
F statistic:	775	346
Y-intercept:	3.55 ± 0.056	3.1 ± 0.108

Table 2: Comparison of curve regressions of Figure 30



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61
20	13	18	19	20																																																									
30	19	24	28	29	29	30																																																							
40	26	32	35	37	38	38	40																																																						
50	33	36	42	45	45	48	49	50																																																					
60	41	47	50	52	53	54	54	57	58	58	60																																																		
70	44	49	56	57	61	61	63	64	66	66	67	67	69	69	69	70																																													
80	51	57	63	68	71	71	75	75	78	78	80																																																		
90	55	62	67	73	76	76	80	82	82	84	84	88	88	88	89	89	89	89	89	90																																											
100	65	70	75	79	81	81	88	91	91	93	93	95	95	97	97	98	98	98	99	99	99	99	100																																								
110	69	77	82	84	86	90	90	92	97	97	101	101	105	105	108	108	108	108	109	109	109	109	109	109	110																																					
120	74	85	85	91	95	97	101	101	106	107	107	109	110	110	110	112	114	114	114	115	115	115	116	116	117	117	117	118	118	118	118	118	120																												
130	83	89	98	99	101	105	105	109	111	114	114	118	118	121	122	122	122	123	123	125	125	125	127	127	127	127	127	129	129	129	129	129	129	129	129	129	129	129	129	129	129	130																			
20	16	18	20																																																											
30	23	24	26	27	28	29	29	29	30																																																				
40	28	31	33	33	36	37	37	39	39	39	40																																																		
50	34	37	40	41	43	44	44	47	47	47	48	48	48	48	49	49	49	50																																											
60	41	44	48	51	51	52	53	55	55	56	57	57	57	58	59	59	59	59	60																																										
70	47	50	53	53	59	60	60	62	62	64	64	64	66	67	67	67	67	68	68	68	68	69	69	69	69	69	69	69	69	69	69	69	69	69	69	70																									
80	55	56	62	62	65	67	67	70	71	71	71	73	73	76	76	76	77	77	77	77	77	77	77	78	78	78	78	78	78	78	78	78	78	78	78	78	78	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	80								
90	60	64	68	68	74	74	76	76	79	79	79	81	81	81	84	84	84	85	85	85	86	86	86	87	87	87	87	87	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88
100	64	67	73	73	78	78	81	88	88	90	90	91	91	92	92	94	94	94	94	95	95	95	95	95	95	95	95	95	96	98	98	98	98	98	98	98	98	98	98	98	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	100					
110	71	75	75	82	82	84	89	89	90	90	94	94	94	94	97	97	97	101	101	101	102	102	102	102	102	102	103	103	103	103	105	105	105	105	105	105	105	105	105	105	105	105	105	105	105	105	105	105	105	105	105	105	105	106	106	106	106	106	106	106	106
120	77	78	86	91	91	92	95	95	97	99	99	101	105	105	105	106	106	106	108	108	108	108	108	109	110	110	110	110	112	112	112	112	113	113	113	113	113	113	113	113	113	113	113	113	113	113	115	115	115	115	115	115	115	115	115	115	115	115	115	115	115
130	78	85	88	94	95	96	96	100	102	102	105	107	107	108	110	110	111	113	113	113	113	114	114	115	115	118	118	118	118	119	119	119	119	120	121	121	121	123	123	123	123	123	123	123	123	123	123	124	124	124	124	125	125	125	125	125	125	125	125	125	125

Figure 30: OneMax for N-Alleles. Blue: Binary search, Red: 10-value allele search



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
20 (blue)	13	18	19	20																											
30 (blue)	19	24	28	29	29	30																									
40 (blue)	26	32	35	37	38	38	40																								
50 (blue)	33	36	42	45	45	48	49	50																							
60 (blue)	41	47	50	52	53	54	54	57	58	58	60																				
70 (blue)	44	49	56	57	61	61	63	64	66	66	67	67	69	69	69	70															
80 (blue)	51	57	63	68	71	71	75	75	78	78	80																				
90 (blue)	55	62	67	73	76	76	80	82	82	84	84	88	88	88	89	89	89	89	89	90											
100 (blue)	65	70	75	79	81	81	88	91	91	93	93	95	95	97	97	98	98	98	99	99	99	99	100								
110 (blue)	69	77	82	84	86	90	90	92	97	97	101	101	105	105	108	108	108	108	109	109	109	109	109	109	110						
120 (blue)	74	85	85	91	95	97	101	101	106	107	107	109	110	110	110	112	114	114	114	115	115	115	116	116	117	117	117	118	118	118	118
130 (blue)	83	89	98	99	101	105	105	109	111	114	114	118	118	121	122	122	122	123	123	125	125	125	127	127	127	127	127	129	129	129	129
140 (blue)	85	93	98	106	107	107	118	120	120	124	126	126	129	129	133	133	133	133	134	134	135	135	135	135	136	136	137	137	137	137	137
150 (blue)	91	102	106	115	119	119	124	129	129	131	134	134	136	136	139	139	140	141	141	141	143	143	144	144	144	147	147	147	147	148	148
20 (red)	13	16	19	20																											
30 (red)	22	24	28	29	29	30																									
40 (red)	26	31	33	35	37	37	38	39	40																						
50 (red)	35	40	44	45	47	47	48	48	50																						
60 (red)	40	45	50	52	53	53	58	59	59	60																					
70 (red)	45	53	59	59	62	65	65	66	67	67	69	69	70																		
80 (red)	49	57	60	63	66	66	71	74	76	76	76	78	78	78	79	79	80														
90 (red)	56	62	67	74	77	77	78	82	82	84	84	86	86	88	88	89	89	90													
100 (red)	62	67	74	77	80	80	83	88	90	91	91	92	92	95	95	97	97	98	98	98	99	99	99	99	99	100					
110 (red)	69	75	81	86	90	90	94	95	99	99	104	104	104	108	108	108	108	109	109	109	109	110									
120 (red)	73	84	89	95	96	96	105	107	107	110	112	112	115	115	116	116	118	118	119	119	119	119	119	120							
130 (red)	82	88	92	97	103	103	110	111	113	113	115	117	119	119	120	120	122	126	126	126	126	127	127	127	127	128	128	128	128	128	130
140 (red)	85	93	106	109	109	116	121	121	125	125	131	131	131	132	132	134	134	134	134	134	135	135	137	137	137	137	138	138	138	138	139
150 (red)	91	102	108	111	115	115	122	124	129	129	131	132	134	134	136	136	138	138	140	140	143	143	143	144	144	145	145	145	145	145	145

	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61
20 (blue)																														
30 (blue)																														
40 (blue)																														
50 (blue)																														
60 (blue)																														
70 (blue)																														
80 (blue)																														
90 (blue)																														
100 (blue)																														
110 (blue)																														
120 (blue)	118	120																												
130 (blue)	129	129	129	129	129	129	129	129	129	129	130																			
140 (blue)	137	138	138	138	138	138	138	138	139	139	139	139	139	139	140															
150 (blue)	148	148	148	148	148	149	149	149	149	149	149	149	149	149	149	149	149	149	149	149	149	149	149	149	149	149	149	149	149	150
20 (red)																														
30 (red)																														
40 (red)																														
50 (red)																														
60 (red)																														
70 (red)																														
80 (red)																														
90 (red)																														
100 (red)																														
110 (red)																														
120 (red)																														
130 (red)																														
140 (red)	139	139	139	139	139	139	139	139	139	139	139	139	139	140																
150 (red)	145	145	146	146	146	147	147	147	147	147	147	147	147	148	148	148	148	148	148	148	148	148	148	149	149	150				

Figure 31: OneMax for N-Alleles. Blue: no correction, Red: Needleman-Wunsch

This binary search fit (red and blue dotted lines in Figure 30) is consistent among genetic operators and standard genetic OneMax implementations [4, 160-163], reflecting similar exponential convergence behaviour. This exponential increase in search complexity against linear increase indicates that Michael O'Neill and Lee Spector's 'Open Issue'[126] and Yampolskiy's principal argument against genetic program inference [125] will persist in this algorithm, even in best-case scenarios.

As the length of the genome increases, the likelihood of detrimental mutation increases. As the ratio of correctly assigned variables increases, the likelihood of beneficial mutation decreases and the likelihood of a necessary allele mutating increases. Therefore, the number of generations leading to convergence generally increases proportionally to the length of the genome for this genetic operator. These results confirm a core limitation of line-level genetic operators: as genome length increases, the chance of necessary alleles mutating rises, driving exponential growth in required generations.

This method is still utilising a cumulative frequency ratio (Figure 15) with large variability and an overflow mechanism for the first genome proportional to the scaffold library length. As gene length, library size and integer distribution should vary heuristically between models, further benchmarking does not appear likely to be utilitarian, however recognising the trend and distribution of outcomes will likely remain consistent regardless of these factors.

Following the indication that source code can still be successfully written when given appropriate heuristic approaches or environments with viable hill climbing search spaces, more heuristic methods were introduced for specific use-cases.

Needleman–Wunsch Alignment for OneMax:

To evaluate the effect of re-alignment on convergence, we repeat the OneMax benchmark using a Needleman–Wunsch–based crossover operator. Testing the impact of crossover alignment on OneMax, identical parameters to the binary search of the previous experiment are given and the crossover mechanism is replaced with the Gene-level Needleman Wunsch algorithm.

Table 3 lists all GA parameters; the only change from the previous OneMax run is the activation of Needleman–Wunsch alignment.

Parameter	Setting
Search	**OneMax Needleman-Wunsch**
Run count	3
Agent count	800
Generation Maximum	100
Nodes	**2**
Bitstring Length	$n = (20,30,40 \ldots,150)$
Mutation Ratio	2%
Injection Ratio	1%
Variable modifier	Mutation 3%, Injection 3%
Starting length	= target length
Needleman–Wunsch Alignment	**ON**

Table 3: genetic algorithm parameters for Needleman-Wunsch experiment

Figure 31 shows that convergence curves with and without alignment coincide, both following $y = 3 \cdot 1.02\hat{}x$: no significant difference is observed from the use of the Needleman-Wunsch algorithm in an elitest-ranking OneMax search. This may be because OneMax alleles are highly similar and the elitist rank-selection preserves only top performers, alignment rarely reorders genes—and thus offers no convergence advantage (Figure 32).

Curve:	$y = 3 * 1.02^x$
R^2 value:	0.947
F statistic:	1017
Y-intercept:	3.55 ± 0.056

Table 4: Curve regression of OneMax results Figure 31

When genes of identical lineage align around a removed gene, they produce one of two outcomes, assuming no further mutation (Figure 32). As alleles are too self-similar to enforce strong alignment at the gene level, including this implementation, there is no significant change in search, both producing the same curve ($y = 3 * 1.02^x$). Selection methods other than elitest ranking may benefit from a correction algorithm like Needleman-Wunsch, however this is not explored in this thesis.

Long Parent	Short Parent	Aligned Short Parent
1A 00000	2A 00000	1A 00000
1B 11111	2B 11111	1B 11111
1C 10101	2C 11111	——
1D 11111	···	1C 11111

Output A:	Output B:
XA 00000	XA 00000
XB 11111	XB 11111
1C 10101	
XD 11111	XC 11111

Figure 32: Demonstration of Needleman-Wunsch producing outputs identical to gene-wise selection.

Given its exponential cost and lack of observed gain, the alignment step is omitted from subsequent. Future implementations may look away from Elitest fitness and may see benefit from this algorithm.

Experiment 2: Unit Test Harness

This section implements a unit-test harness to validate GLGPPS's ability to generate functionally correct code for simple mathematical problems. A pair of simple coding exercises are implemented to demonstrate functionality and completion given simple problem spaces: Hill climbing towards pi and three-value addition.

Hill Climbing

This experiment I a test to converge on an irrational number, allowing an exploration of speed, accuracy and algorithmic conjugation of GLGPPS with a restricted maths library. Agents are scored on performance and bred to select for the best performance. This scenario supports gradient descent as a search, as a single maximum exists in a clean gradient fitness explicit landscape.

When generating before unused variable removal is implemented, this algorithm proved to be susceptible to genetic bloat (large genetic sequences and output code which could be achieved with shorter statements), if-statements which can never trigger, new variables which are never used and low value manipulation of values which could be applied in a single operation. This is not uncommon to linear-genetic program-synthesis algorithms [84], [125].

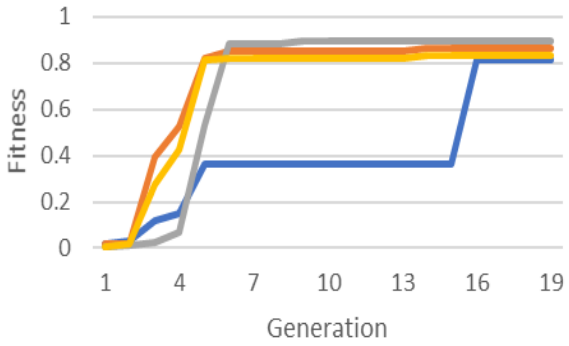


Figure 33: Fitness over generations:

5 simulations demonstrating rapid convergence, 100 agents per generation.

Initial tests of this algorithm were against explicit fitness functions and did not implement templating. They were run to complete mathematical and algorithmic tests of increasing complexity, successfully with rapid convergence where the solution was solvable, though not outside the expected outcomes of other GP systems.

Figure 33 gives an example of this convergence, demonstrating fitness per generation on a series of experiments attempting to solve $f(x) = Pi$. In this experiment the fitness was only rewarded up to 0.9 for result accuracy, with the 0.1 additional score overhead allocated for shorter code solutions beyond perfect completion.

Figure 34 demonstrates an example output code stub for the algorithm from these experiments using a simple math scaffold library (Figure 37). This code block is an example of the entire generated code including its wrapper, as emitted and reflected by the compile-reflection system.

This experiment took a form of Unit Test [164], [165] against mathematical functions, generally irrational numbers and standard regression problems. This approach allows for the simple construction by humans, of a test-driven-design lead automatic collaborator.

These results confirm that scaffold design critically impacts both convergence speed and output readability. Currently this is a manual process, but these tests demonstrate that an automatic system which can isolate which lines in a library are likely to be useful and assign a weight to them. This would allow a system to automatically determine its own use case and select an appropriate library, functionality which would be critical in potential integration as a collaborative programmer.

```
using System;
using System.IO;
namespace RoslynCore
{
    public static class Helper
    {
        public static double GeneratedCode(double output)
        {
            output += Math.Sin(83);
            double a = 22;
            output = 36 / output;
            output += Math.Pow(11, output);
            output += Math.Sin(output);
            output += Math.Sin(output);
            output += Math.Sin(output);
            if (output > a)
            {
                output = 8 * output;
                output = output * 27;
            }
            return output;
        }
    }
}
```

Figure 34: Example code output attempting to solve:

$$f(x) = Pi$$

Tests were then executed to assess the impact of mutation, insertion and removal on a species which had no variance in genetic length. This explored a scenario in which 1000 agents were executed for 100 generations. The results clearly demonstrate the significance of tuning mutation, insertion, and removal ratios, with both the highest and lowest mutation ratios providing the average lowest scoring results. We can anticipate that the optimal ratios are likely strongly dependant on the search space.

The results (Figure 35) demonstrate that a combination of mutation, removal and insertion outperforms each element independently.

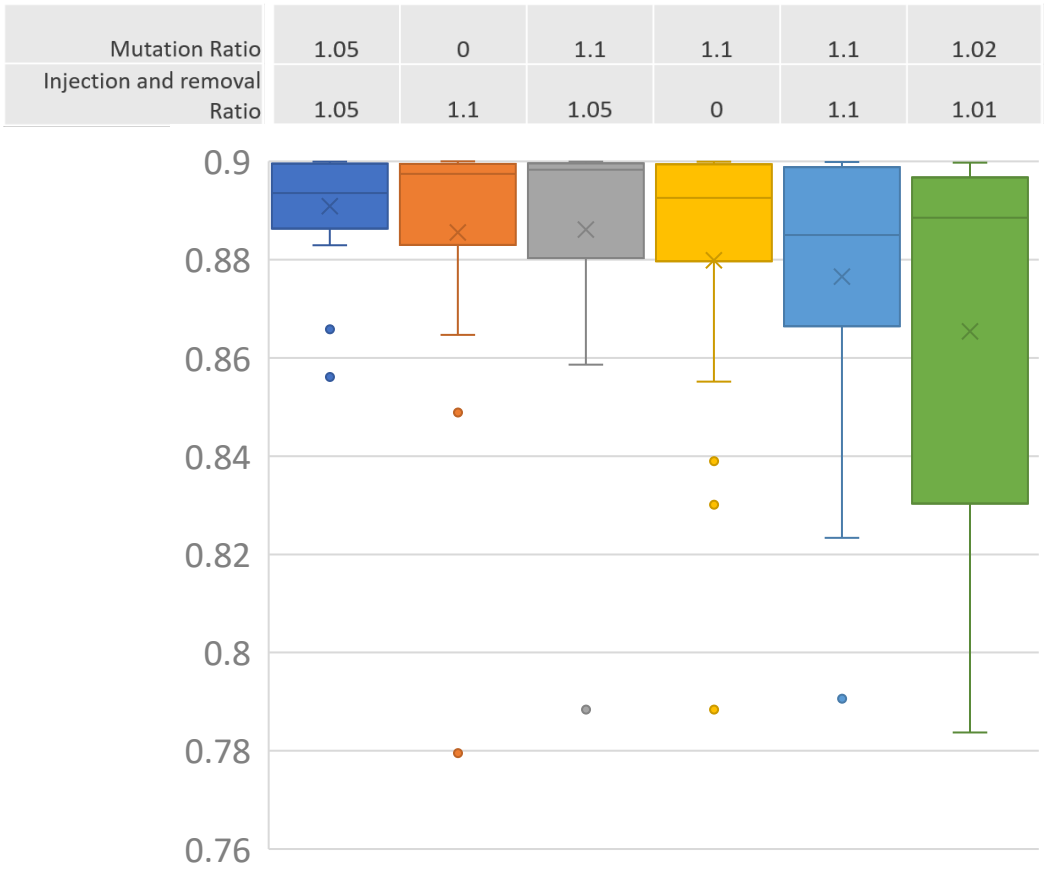


Figure 35: Fitness after 20 generations, demonstrating the impact of mutation.

Following this, a stress test was run, executing 10,000 agent delegates each with 300 lines of code in under 50ms to execute. It should be noted that stress test was against the execution of the delegates, not against the generation of agents, which varies on method of generation and length of agent.

This prototype was published [166] as a C#.net core auto-coder without a templating mechanism. It was clear the algorithm did not necessitate C#, so a variable library functionality was written for the algorithm, which allowed it to swap between languages and output a text file directly.

Simple Code Exercise Solving

Moving beyond standard hill climbing towards a standard variable, the algorithm is submitted to a series of unsupervised, simple coding tests to resolve simple maths problems using a source code format.

The AI was given access to a short, simple maths scaffold library (Figure 37), with 500 agents per generation, using an elitist selection algorithm, breeding only between the top 10 scoring agents per generation. Fitness is determined by successful completions of a set of 1000 unit tests with input parameters run against a known working solution. This is likely to be a detrimental ranking system for the search space [167], but offers a simple method to validate the plausibility of the model as an auto-coder for arbitrary coding tasks.

This approach makes use of the dynamic compilation framework and directly tests compiled code to see if the output matches the output of a compiled, correct solution, live. This utilises the convert, compile, commit, delegate loop (Figure 28), using dynamically generated DLLs.

Simple Addition

The AI was tested against the task of resolving the sum of three input values (Figure 36), initially as integers between 1 and 10. The unit test was a simple one-line expression, though the scaffold library in use (Figure 37) did not give solutions for operations with more than two values, so would require a multiple line solution to resolve.

```
public static double Test3(double x, double y, double z)
{
    return x + y + z;
}
```

Figure 36: Three input sum unit test

This experiment would consistently resolve in either one or two generations. An example of a complete solution before unused variable removal can be seen in Figure 38, which successfully resolves in the first 3 lines, but holds an additional 5-line non-coding region. An interesting observation is the use of the return statement: this statement appears frequently and separates clear coding and non-coding regions in experiments which resolve in a shorter number of generations.

This demonstrates that GLGPPS can locate solutions to simple problems in this environment, though the two-generation solutions generally resolved due to 0 being one of the input integers, allowing the first generation to succeed 10% of the time with an intermediate solution which only returning the sum of two of the input values.

```
//Create new var
    new List<string> { "double ","2"," = 0;", "0"},
    new List<string> { "double ","2"," = output;", "0"},
    new List<string> { "double ","2"," = ","0",";", "0"},
 //terminate block
    new List<string> { "}" , "-1"},
    new List<string> { "}" , "-1"},
//Assign dynamic value to new value
    new List<string> { "double ", "2", " = ", "0", ";" , "0"},
//Assign dynamic value to dynamic value
    new List<string> { "double ", "2", " = ", "1", ";" , "0"},
//assign output to dynamic variable
    new List<string> { "output =", "1", ";" , "0"},
//IF
    new List<string> { "if(", "1",">", "1", "){", "+1"},
    new List<string> { "if(", "1","<", "1", "){", "+1"},
    new List<string> { "if(", "1","==", "1", "){", "+1"},
    new List<string> { "if(", "1","==", "0", "){", "+1"},
    new List<string> { "if(", "1","!=", "1", "){", "+1"},
    new List<string> { "if(", "1","!=", "0", "){", "+1"},
///ELSE
    new List<string> { "} else if(", "1"," > ", "1", "){", "0"},
    new List<string> { "} else if(", "1"," < ", "1", "){", "0"},
    new List<string> { "} else if(", "1"," > ", "0", "){", "0"},
    new List<string> { "} else if(", "1"," < ", "0", "){", "0"},
    new List<string> { "} else if(", "1","!= ", "1", "){", "0"},
    new List<string> { "} else if(", "1","!= ", "0", "){", "0"},
//RETURN
    new List<string> { "return ", "1", ";", "0" },
///GENERIC MATH OPERATORS
    new List<string> { "1", " += " ,"0",";", "0"},
    new List<string> { "1", " += ","1",";", "0"},
    new List<string> { "1", " += ","1",";", "0"},
    new List<string> { "1", " -= ","1",";", "0"},
    new List<string> { "1", " -= ","1",";", "0"},
    new List<string> { "1", " =  ","1"," * ","0",";", "0"},
    new List<string> { "1", " += Math.Sin(","0",");", "0"},
    new List<string> { "1", " += Math.Cos(","0",");", "0"},
    new List<string> { "1", " =  Math.Pow(", "1", ",", "0",");", "0"},
    new List<string> { "1", " += Math.Pow(","0", ", ", "1", ");", "0"},
    new List<string> { "1", " += Math.Pow(","1", ", ", "1", ");", "0"}
```

Figure 37: Example simple maths scaffold library

```
inputA += inputB;
inputA += inputC;
return inputA;
double b = 0;
inputA += Math.Pow(39, inputB);
inputB -= output;
double c = inputA;
return output;
```

Figure 38: Successful "Simple Addition" task completion

These unit-test results demonstrate GLGPPS's rapid problem-solving capacity in constrained domains, but highlight the need for automated scaffold pruning to mitigate code bloat. Scaffold design critically impacts both convergence speed and output readability. GLGPPS locates correct solutions rapidly in simple domains but suffers scaffold-induced bloat and unused-variable overhead.

Experiment 3: Helmuth & Spector's Benchmarks:

We assess GLGPPS on a subset of Helmuth & Spector's asynchronous I/O benchmarks, comparing completion rates and code legibility against published G3P results [168]. This is the first benchmark suite and, at the time of writing this thesis, the only standard for direct analysis of source code outputs for genetic program synthesis.

As program synthesis is a developing field, very few implementations with documented benchmarks against a standardised framework exist. Helmuth and Spector's benchmark suite aimed to address this, but adoption is still limited. Benchmarking will be compared to Spector and Helmuth's results where appropriate with an emphasis on legibility. Helmuth and Spector suggest benchmarking primarily on completion; however, this thesis looks to analyse human comprehension of code using quantitative frames.

For these experiments, GLGPPS utilises a heuristic program synthesis: the templating mechanism does not currently operate with an automatic scaffold pruning framework, so models use an arbitrary longer library (Figure 37), and a specific heuristic sample of this library derived for common solutions to the specified problem. Notably, the benchmark G3P model also uses a human-selected library of nodes for construction.

This version of GLGPPS is also not designed for string manipulation: strings may be returned but string manipulation is not assessed. String manipulation as a functional library however can be added to GLGPPS, including, as suggested in Helmuth & Spector's initial paper, addressing regression towards a correct string output using "*a Levenshtein distance* [169] *(a measure of string edit distance) as the fitness function*", however this remains out of the scope of this project.

Gene injection and removal algorithms are active in these tests, so number of final genes may differ from starting genes.

For all experiments:

Agent population:	800
Mutation Ratio:	2%
Injection Ratio:	1%
Success modifier:	Mutation: 3%, Injection: 3%
Failure Modifier:	Reset to: (Mutation: 5%, Injection: 3%)
Fitness Function:	# accurate matches of input to output for pre-solved set

Table 5: genetic algorithm parameters for all GLGPPS benchmark experiments

Fitness for the following experiments is given by the number of correct outputs from the Helmuth and Spector benchmark Suite – a library of inputs and expected outputs for a range of computing tasks.

Library size: a generic maths "larger library" is provided for all experiments, and a specific manually selected "heuristic" library is used in "heuristic models". For the "heuristic models", lines of code which can be demonstrated to complete the problem are selected as the library.

Measures for "lines of source code" include the wrapper function not included in code excerpts. Graphs for completion for each test can be found in Appendix B.

Results analysed are taken from the first working solution in any experiment which solves all given tests.

IO

We evaluate GLGPPS on the simplest I/O task: returning the sum of two inputs (an integer and a float). A heuristic library with only addition, return assignment scaffolds was also tested.

Parameter	Setting
Search	**IO**
Library Size	**40 (Larger library) & 4 (Heuristic)**
No. starting genes	10

Table 6: starting parameters for 'IO' benchmark experiment

GLGPPS generated a successful solution in the first generation, on both large library and heuristic models (Figure 39).

```
inputB += inputC;
output = inputB;
return output;
```

Figure 39: GLGPPS first solution for "IO" Benchmark test

Table 7 compares cognitive complexity, maintainability index, cyclomatic complexity, and code-length against G3P. Both systems yield equivalent functional and structural metrics—differing only by one extra line in GLGPPS due to brace syntax in C#.

	G3P	GLGPPS	Difference
Cognitive complexity:	0%	0%	0%
Maintainability Index:	76	75	-1
Cyclomatic Complexity:	1	1	0
Lines of Source code:	5	6	+1
Lines of Executable code:	4	4	0

Table 7: Readability metrics comparison of 'IO' experiment results for G3P and GLGGPS

Behaviour-wise, results between GLGPPS and G3P are identical. G3P derives a higher score despite identical functionality due to Python's syntax not including terminating bracers. Despite using different scaffold libraries, GLGPPS matches G3P's cognitive and cyclomatic complexity on this trivial task - the +1 line in GLGPPS stems from C# block delimiters; G3P's Python output omits braces. This result confirms that, for this simple I/O tasks, scaffold design does not impair readability or conciseness—validating the minimal overhead of our templating approach.

Median

This section evaluates GLGPPS on the 'Median' I/O task, which returns the middle of three inputs. It also explores variation in readability metrics and their applictionin between benchmark and tested algorithms.

Parameter	Setting
Search	**Median**
Generation Maximum	100
Nodes	**40 (Larger library) & 5 (Heuristic)**
No. starting genes	10

Table 8: parameter settings for 'Median' benchmark experiment

With the full library, GLGPPS failed to converge within 100 generations. Using the heuristic library, a correct solution emerged at generation 78.

With a heuristic sample, a sub-set of the scaffold library (Figure 37) with only comparative and return statements, a complete solution was synthesized after 78 generations (Figure 40: GLGPPS first solution for "Median" Benchmark test).

```
output = inputA;
if (inputB >= inputC){
	if ((inputC >= output) && (output >= inputA))
	{
		return inputC;
	}
	else if (inputA >= inputB)
	{
		return inputB;
	}
}
else if (inputB >= output)
{
	return inputB;
}
else if ((inputA >= inputC) && (output > inputB))
{
	return inputC;
}
return output;
```

Figure 40: GLGPPS first solution for "Median" Benchmark test

	G3P	GLGPPS	Difference
Cognitive complexity:	40%	80%	+40%
Maintainability Index:	58	60	+2
Cyclomatic Complexity:	3	8	+8
Lines of Source code:	19	25	+6
Lines of Executable code:	12	11	-1

Table 9: Readability metrics comparison of 'Median' experiment results for G3P and GLGGPS

The comparison of results between G3P and GLGPPS suggest substantial improvements to human comprehension of code in the G3P model. This result is worth analysing as it raises a major contention in human comprehension against automatic comprehension measures.

```
def evolve(in0, i0, i1, i2):
    res0 = int()
    i0 = max(min(max(in0, in2), divInt(in1, +min(abs(i1),int(3.0)))),
min(in0, divInt(min(mod(divInt(in2, i2), i0), (max(i2, in0) * min((in0
- divInt(-abs(i1), max(i1, i1))),+abs(in0)))), +min(abs(i1),
int(3.0)))))
    b1 = b1
    b1 = b2
    while not False:
        res0 = i0
        if loopBreak > loopBreakConst or stop.value:
            break
    loopBreak += 1
    return res0
```

Figure 41: G3P example solution for "Median" Benchmark test

The G3P solution (Figure 41) demonstrates one complex single line solution and a series of statements and assignments which have no functionality. This can be generalised into a 2-line solution Figure 42.

```
res0 = max(min(max(in0, in2), divInt(in1, +min(abs(i1),int(3.0)))),
min(in0, divInt(min(mod(divInt(in2, i2), i0), (max(i2, in0) * min((in0
- divInt(-abs(i1), max(i1, i1))),+abs(in0)))), +min(abs(i1),
int(3.0)))))
return res0
```

Figure 42: G3P after removal of functionally redundant code

This solution would score a maintainability index of 76 and cyclomatic complexity of 1, implying a more conventionally readable solution with these measures however, this solution demonstrates substantial horizontal density, in a single line: min, mod, max, abs and integer conversion of a new float are all called with complex logic nesting.

A simple modification of the linting algorithm was run on the G3P solution for this problem, identifying some improved readability but retaining substantial line density.

	G3P with lint	G3P	GLGPPS	Difference
Max Horiz. density:	241	241	52	-189
Operands:	18	33	21	-12
Operators:	26	38	20	-18
Brackets & Bracers:	46	48	27	-21
Max. logic nesting:	9	9	4	-5

Table 10: Horizontal density analysis of 'Median' Benchmark results

It is worth addressing that significant horizontal density without necessity does not follow good coding conventions [170]; a human programmer producing clean code should atomise new variable assignments which minimise the functionality of an individual operation. Horizontal density is therefore not clearly quantified by automatic complexity calculations which derive line count but not horizontal as part of the quantitative metric.

Small or Large

This algorithm returns:

This experiment evaluates GLGPPS on a threshold-classification task: return “small” if $x < 1000$, “large” if $x \geq 2000$, and “” otherwise. Because the baseline implementation cannot construct literal strings, the tokens “small” and “large” were pre-injected into the scaffold library as return-value constants.

Table 11 lists the genetic-algorithm parameters. Two scaffold libraries were tested: a full “larger” library of 40 nodes and a minimal “heuristic” library of 5 nodes containing only

assignment, comparison, and return templates. All runs used an elitist selection with gene injection/removal enabled.

Parameter	Setting
Search	**Small or Large**
Generation Maximum	100
Nodes	**40 (Larger library) & 5 (Heuristic)**
No. starting genes	10

Table 11: parameter settings for 'Small or Large' benchmark experiment

Given these parameters, this algorithm failed to identify a complete solution, however many partial completions were derived.

Models both with and without mathematical derivation techniques both became trapped in the local maxima of using a default value (of zero) as an upper and lower boundary.

Support values of 1000 and 2000 are added as part of the library for this problem, with them the algorithm converges consistently (Figure 43). The G3P algorithm utilises the same heuristic mechanism for this problem. Results converge into uniformly better scoring outputs than the benchmark, however these results do not attempt to converge numbers 1000 or 2000.

```
inputB = 1000;
inputC = 2000;
if (inputA < inputB)
{
    return -1;
}
else if (inputA >= inputC)
{
    return 1;
}
return output;
```

Figure 43: GLGPPS first solution for "SMALL OR LARGE" Benchmark test

Table 12 compares readability and complexity metrics against G3P. GLGPPS achieves lower cognitive complexity (20 % vs. 62 %), lower cyclomatic complexity (2 vs. 8), and dramatically reduced horizontal density, at the expense of four extra scaffold lines.

These results demonstrate that embedding problem-specific constants in the scaffold library is essential for convergence. GLGPPS's templated output is significantly more readable than G3P's, as shown by a 42 % reduction in cognitive complexity and an 189-point drop in maximum horizontal density.

	G3P	GLGPPS	Difference
Cognitive complexity:	62%	20%	-42%
Maintainability Index:	57	62	+5
Cyclomatic Complexity:	8	2	-6
Lines of Source code:	13	17	+4
Lines of Executable code:	12	8	-4
Max Horiz. density:	131	26	-105
Operands:	36	11	-15
Operators:	55	9	-46
Brackets & Bracers:	74	8	-66
Max. logic nesting:	9	1	-8

Table 12: Maintainability & cognitive complexity statistics for "Small or Large" experiment

Sum of Squares

This experiment measures GLGPPS's ability to compute the sum of squares for all integers in [1, n], a task with no smooth fitness gradient and requiring a multi-line loop solution.

Parameter	Setting
Search	**Small or Large**
Generation Maximum	50
Nodes	**40 (Larger library) & 6 (Heuristic)**
No. starting genes	10

Table 13: parameter settings for 'Sum of Squares' benchmark experiment

This is a problem with a specific, multiple-line solution which does not provide a clear route for gradient descent. The results for this search align with this type of search: a high failure ratio with a range of low scoring local maxima, with a sudden jump to a completion in a single generation.

This test failed 8 complete runs of 50 generations (421 total generations) before a successful completion (Figure 44) was found.

```
for (double a = 0; a < inputA; a++)
{
    inputC += Math.Pow(a, 2);
    output = inputC;
    output += Math.Pow(inputA, 2);
}
return output;
```

Figure 44: GLGPPS first solution for "SUM OF SQUARES"

The sudden jump to a solution after many failures reflects the non-gradient nature of this loop-based task. GLGPPS's scaffolded code is both shorter and structurally simpler than G3P's, reducing horizontal density by 142 points and cutting cyclomatic complexity in half. These results validate that templated, multi-line constructs can yield more

maintainable arithmetic algorithms, even when convergence is sporadic and reliant on heuristic library design.

	G3P	GLGPPS	Difference
Cognitive complexity:	2 (25%)	1 (10%)	-1 (15%)
Maintainability Index:	61	68	+7
Cyclomatic Complexity:	3	2	-1
Lines of Source code:	11	10	-1
Lines of Executable code:	11	6	-5
Max Horiz. density:	177	35	-142
Operands:	37	14	-23
Operators:	56	9	-47
Brackets & Bracers:	96	8	-88
Max. logic nesting:	14	3	-11

Table 14: Maintainability & cognitive complexity statistics for "Sum of Squares" experiment

Results

Results of GLGPPS appear comparable to existing benchmarks for GP-derived program synthesis algorithms in terms of completion for the given tests.

It should be noted that GLGPPS does not currently have a method for automatic scaffold selection, hence the Large-Library and Hueristic models being separate, but results indicate comparable outcomes in terms of generated functionality to G3P.

Experiment	Attempts	Generations to solution	Maintainability Inex	Cyclomatic Complexity	Cognitive Complexity
IO	1	1	62	2	0 (0%)
Median	1	40	61	11	8 (80%)
Small or Large	1	22	62	2	2 (20%)
Sum of Squares	8	21	68	2	1 (10%)

Table 15: Summary of GLGPPS Heuristic model benchmark solutions

GLGPPS produces many lines of code, with clearly atomised behaviours, opposed to G3P solutions, which generate many complex single-line statements.

There may be scenarios where complex single-line searches are preferred but for the extents of this experiment, the general maintainability and comprehension of generated code is improved in GLGPPS.

One of the substantial benefits to the many-line approach is improved compatibility with linters and static code analysis tools: a simple unused-variable removal algorithm

demonstrated substantial improvement to lines of code and maintainability of code in the output file. This would not be as impactful for algorithms which produce complex single-line solutions which introduce many new operands in the same line.

Case Study: Sum of Squares

To demonstrate how GLGPPS operates and as an example of output comparison to G3P, "Sum of Squares" is selected as a case study. This is due to be being ranked as a more difficult problem to resolve [168], with similar success rates for both algorithms and a substantial demonstration of the change in output due to the unused variable removal in the case of GLGPPS.

Analysing the impact of the removal of unused variables (Figure 45), this algorithm reduces the lines of executable code by 60% and total lines of code by 45% in this solution, with no modification to functionality.

```
for (double a = 0; a < inputA &&
itterations < maxItterations; a++,
itterations++)
{
    double b = output;
    double c = inputA;
    inputC += Math.Pow(a, 2);
    inputB += Math.Pow(output, 2);
    output = Math.Pow(a, 2);
    output = inputC;
    inputB = Math.Pow(inputC, 2);
    output = inputC;
    inputB = Math.Pow(a, 2);
    output += Math.Pow(inputA, 2);
}
return output;
```

```
for (double a = 0; a < inputA; a++)
{
    inputC += Math.Pow(a, 2);
    output = inputC;
    output += Math.Pow(inputA, 2);
}
return output;
```

Figure 45: GLGPPS first solution for "SUM OF SQUARES" Benchmark test before (Left) and after (right) unused-variable removal

	Before unused variable removal	After
Cognitive complexity:	2 (20%)	1 (10%)
Maintainability Index:	72	68
Cyclomatic Complexity:	3	2
Lines of Source code:	18	10
Lines of Executable code:	15	6

Table 16: Readability metrics comparison of 'Sum of Squares' experiment results before and after linter

This removal process uniformly improves maintainability and complexity measures against the initial output.

Figure 46 demonstrates the G3P solution for the same benchmark. The same issues raised in this section are present across the benchmark solutions. *See Forstenlechner's thesis* [168] *for a complete library of G3P solutions.*

This solution distinctly creates complex statements with many in-line operators for a single procedure. Due to the length of these statements, there are three lines of code which break PEP8 conventions for line length [171], with lines of 178, 147 and 95 characters. This format of construction is endemic in Grammatical Evolution algorithms.

G3P solutions also produce a large number of magic numbers [170], which ideally should be replaced with (named) constants, or at least emphasise evolving solutions which create named variables and use said variable rather than evolving magic numbers. This is especially pressing for solutions which are not discovered with intent: evolving magic numbers often creates solutions which solve a given test or series but solve with a maths formula defined through a representation of highly specific numeric values which have been located through gradient descent.

Comparing GLGPPS and G3P in these terms, we can see that enforcing a limit on the number of operands in a single line helps to remove the likelihood of horizontal bloat in a single expression without compromising either comprehension or completion. We can also see improvements of encapsulation of boundary conditions.

GLGPPS in the Helmuth & Spector experiments also prevent the use or creation of numbers unless a direct instantiation of a value into a new variable or the use of common values (such as 0, 1 or 2) inside single line operations ought to also be applied, which supports readability against the benchmark.

These principles, when applied, may lead to more vertical solutions, however the vertical bloat is suppressed more actively due to the removal of unused variables. Notably, horizontal bloat is not mitigated through this algorithm, but an alternative linter may support cleaner code.

```
def evolve(in0, i0, i1, i2):
    res0 = int()
    res0 = divInt(( +max(in0, abs(int(6.0))) + abs(abs(max(i2,
(abs(max(i1, ( divInt(+( in0 + ( +min(abs(in0), int(3.0)) +
divInt(in0,i2) ) ),i2) * in0 ))) * in0 )))) ),int(6.0))
    i0 -= i0
    if i2 > min(( ( +( i2 - ( mod(+( +in0 + in0 ),+( int(979.0) +
int(95880.0) )) * int(640.0) ) ) + -i2 ) + ( +int(0.0) - i2 ) ),
abs(int(97.0))):
        i0 -= abs(int(6.0))
    else:
        res0 = divInt(( abs(int(970497.0)) + ( int(6.0) +
int(511889.0) ) ),abs(abs(int(6.0))))
    return res0
```

Figure 46: G3P solution for "SUM OF SQUARES" Benchmark test, for comparison, source: [168]

For this problem, against the benchmark G3P model, we can conclude the following:

1. GLGPPS produces more maintainable code:
    a. less likely to generate magic numbers
    b. less likely to breach length guidelines
    c. less subject to horizontal bloat
    d. more compatible with linters to support code cleanliness
2. GLGPPS can converge working solutions to benchmark problems in a similar number of searches.

Beyond GLGPPS, one significant finding in this research is that current maintainability calculators which emphasize complexity as an expression of indentation depth or count number rather than complexity of operators will calculate complexity of a solution which is not representative of the difficulty for a human to interpret an algorithm's behaviour.

These algorithms are not designed for program synthesis, assuming most conventions are already followed to guide a human towards a lower cognitive complexity. Program synthesis should remain critical on how complexity is being quantified, especially if automatic maintainability calculation is brought into the fitness function of an algorithm.

Experiment 4: Artificial Life

Exploring a synchronous task, to test GLGPPS's capacity for human interaction and implicit fitness (no explicit fitness function but a fitness requirement imposed by the environment itself), a loosely guided, rapid adaption scenario is presented: an artificial life simulation.

A series of experiments were applied using the GLGPPS algorithm, including a Template-Based variant and a Needleman-Wunsch variant of GLGPPS, as a behavioural controller for a virtual species in a virtual environment. These experiments analysed the automatic evolution of behavioural nuance and ecological niche under different environmental conditions and population limitations. This implementation made use of multiple function implementation: functions with different behavioural controls which are selected dependant on a template-based mechanism [123] and evolve independently.

A series of sub-experiments explores implicit fitness with and without human interaction. This will explore if, following biologically inspired theory on complexifying genetics, an implicit fitness environment might modify the evolutionary procedure.

Agents in this experiment evolved as when a pair of agents moved near each other, and one agent had reached a breeding countdown timer set first at birth and re-set after successfully breeding.

This sub-experiment also analysed human interactions with the virtual species, analysing the role of human co-evolution with the artificial species in a Mixed-Reality context.

Sub-Experiment 4.1: GLGPPS behaviour controller for Artificial Life

The first sub-experiment ran GLGPPS, without templates. Several sub-experiments were run in this category, exploring: the underlying evolutionary mechanisms the agents exhibited the dynamics of the system and the speed of adaptation to changing environments.

Disabling the templating mechanism for these tests simply meant running the algorithm using a single function, without a hard-coded template linking together a sequence of functions in the behavioural controller.

The mutation, insertion and removal ratios are carried forward from previous experiments.

Agent population:	800
Mutation Ratio:	2%
Injection Ratio:	1%
Fitness Function:	(Implicit) survival

Table 17: parameter settings for 'Artificial Life' experiment

In the first generation, all agents are born with the same generic, hard coded genetic sequence. This sequence constructs its phenotype into a simple finite state machine

(Figure 47) which simply instructs the agent to follow simple behaviours depending on the values passed in from the physics engine about that agent's local environment. A hard coded starting genotype is used to minimise population fall off when a fully random population is generated on the first generation.

```
0 ,  0,  0,  0,  0
5 ,  4,  0,  0,  0
12,  1, 30,  0,  0
6 ,  0,  1,  0,  0
39,  0,100,  0,  0
5 ,  2,  0,  0,  0
39,  2,100,  0,  0
5 ,  3,  0,  0,  0
      (auto-terminate)
```

```
float a = 0;
target = closestCluster;
if("distance" < "30" ){
    target = Lerp(agentPosition,closestPred,-1);
} else if("hunger" < "100" ){
    target = closestFood;
} else if("agentAge" < "100" ){
    target = closestBreedingTarget
}
```

Figure 47: hard coded first generation.
Left: Genetic matrix. Right: generated phenotype using genetic matrix.

This default behaviour prioritises predator avoidance, moving in the opposite direction to a predator if they are within a detection radius of that predator. It then checks its hunger value, prioritising explicit survival, it heads in the direction of the nearest available food source. If it is not in imminent threat of termination, it will either attempt to locate a breeding partner if it is old enough to push a breeding request, or it will simply herd by moving towards the centre of the nearest K-means cluster.

This sub-experiment demonstrated the principle of Occam's razor, as agents tend towards either a direct one-line solution to survival or a bloated solution which prioritises single-line behaviour, using the size of the genome to protect the behaviour from mutation and removal procedures.

Sub-Experiment 4.2: TB-GLGPPS

A template mechanism was produced for agents in this experiment, moving the agents' genetic data structure from 2D to 3D to allow multiple functions to operate on each agent simultaneously. This operates as a series of functions which are generated for each agent.

A generic, hard-coded template is created, which controls which, when and to what extent each of these functions activate and what power they have over the steering mechanism. This is the 'template' of the 'template-based' mechanism.

Each agent in the following Template-Based experiments had 6 chromosomes, where the first 5 chromosomes are used to construct complete functions in the same way the GLGPPS experiments operate: outputting a desired location in 3D space which is converted into a direction. The final chromosome is used as a weighting matrix for the templating mechanism, derived directly from the first two genes only.

The first gene of the final chromosome is read directly into the weighting matrix from the genetic sequence itself and therefore is subject to crossover and mutation. The weighting matrix is used in the Augmented Finite State Machine (AFSM) element of the Template-Based operator (Figure 48) to control which genes are utilised at any one time

and what there weighting is towards the final target location. This is utilised in the agents' steering mechanisms, allowing evolution to optimise when which triggers should activate based on agents' sensors and states as well as what to do when those triggers are active.

Figure 48:Psuedocode of Augmented Finite State Machine operator in TB-GLGPPS templating mechanism

```
1.  gene_data = chromosome[6].gene[0]

2.  if Predator_distance < gene_data[0] then
3.      targets.add( behaviour[0])
4.  if front_sensor_hit and (hit.distance < gene_data[1]) then
5.      targets.add( behaviour[1])
6.  if hunger > gene_data[2] then
7.      targets.add( behaviour[2])
8.  else if breedCount_Down > gene_data[3] then
9.      targets.add( behaviour[3])
10. else
11.     targets.add( behaviour[4])

12. target = weightedAverage(targets)
```

This algorithm allows us to explore a more strongly guided evolutionary approach, combining a broad human template that the program synthesis algorithm constructs around.

Generally, this species would minimise values for all behavioural scenarios of the template except for one and replace the behaviour of that function with a one-line solution.

Scenarios were presented which forced more complex behaviour, such as the introduction of a predator and strict limitations on breeding, which did produce more complex behaviours, but behaviour would trend towards the simplest solution that could survive, minimising function loss from mutation while subsisting. Population dynamics would see a rapid loss in population followed by a stabilisation around a simple effective phenotype (Figure 49).

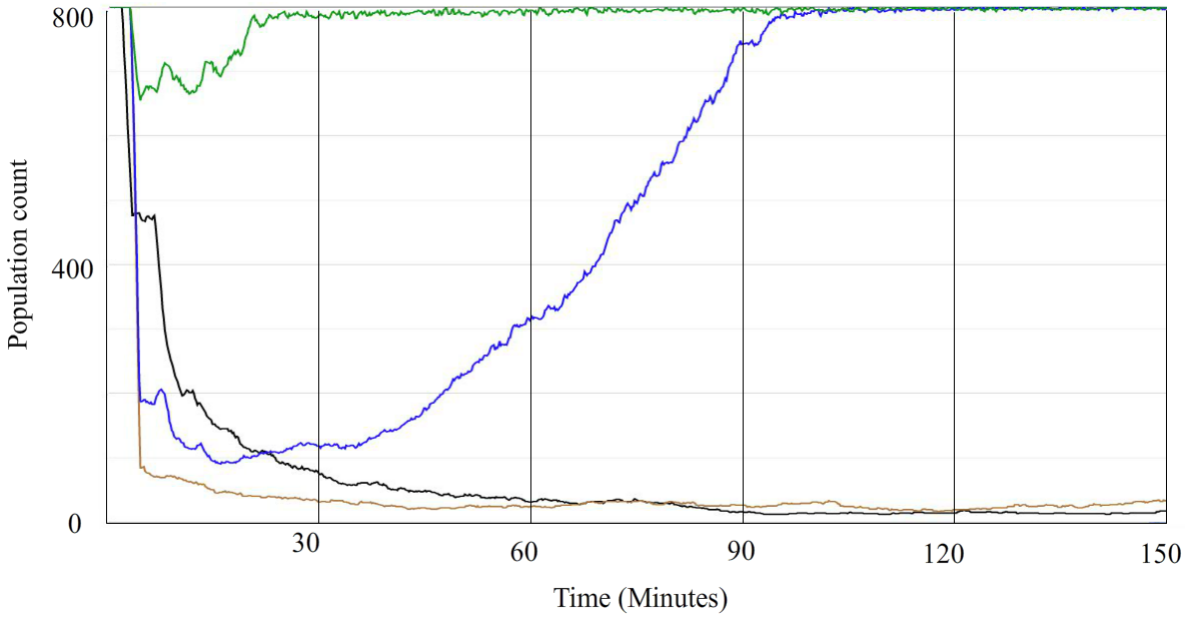


Figure 49: Comparison of population dynamics after initialisation:
TB-GLGPPS with modified starting gene (green)
TB-GLGPPS with expanded spawning region (black)
TB-GLGPPS with small spawning region (brown)
GLGPPS with small spawning region (blue)

Experiment 5: SuperCollider Collaborative Music Generation

The GLGPPS algorithm as a source code auto-coder, without templating, was applied and published as an AI collaborator for a live music performance [172], [173] to an audience of 36 participants. This model modified the original C# scaffolds to work with SuperCollider – an object-oriented music programming language and took human input to evolve programs that evolved sounds. This effectively converted the algorithm to generate SuperCollider code.

This project output code to a program called 'Utopia' [174], a Just-in-time, live-compilation audio generation programming interface for collaborative live music coding performances, traditionally between human practitioners. This programming interface allows users to program collaboratively together to generate music from mathematical formulae, which can be interpreted in a C-based code format.

The GLGPPS algorithm utilised this format. It was given a range of simple mathematical function calls and object instantiation code blocks as its scaffolds, allowing the system to utilise, generate and manipulate mathematical representations of waveforms through nested function calls and statement construction.

This algorithm generates supercollider code on the host C# .net Core server, rather than in Supercollider itself. To output to the client, this algorithm makes use of its webpage output, transmitting across a local network to display to a client device which is running an instance of Utopia. A standalone application was used to copy the output from a webpage to the application, live. An alternative implementation was also generated which inherited directly from the base C# project. Both implementations copied code from the web interface to the Utopia environment at typing speed to produce an illusion of a human-like programmer.

In generating audio, a fitness function becomes necessary, which is where the human audience element is introduced. This algorithm was designed to take human input as a fitness function, applying a score relating to audio preference of sounds. The genetic algorithms for each result were ordered by the score received from the human participants.

To evolve the ordered list of algorithms, a traditional elitist ranking evolution [175] approach was implemented. The algorithm successfully and consistently demonstrated an ability to converge towards consistent sound styles when reinforced towards them by human feedback. This was a noticeable change in generated tone within a single generation, which was consistent across several experiments. This persisted across experiments with smaller population sizes and for shorter sound sample durations.

Sounds generated during prototyping were often multi-tone sounds, compromising of both a high pitch and a low pitch frequency, generating a more vocal sounding hum when optimising against the preference of an audience.

One of the sub-experiments inverted the order of the list, effectively inverting the search space to create a sound with the weakest human preference. This approach generated either no sound or very sudden, very loud sounds. Demonstrating this form of convergence against the human selective pressure for an intuitive inverted preference

gives evidence that the algorithm functions as intended on the sample sizes utilised against human selection for this medium.

The number of agents in a generation was reduced to 6 and the sample length was reduced to 10 seconds, making a full generational sample last a single minute. The size and duration were reduced in preparation for a public facing experiment. Models with this number demonstrated effective convergence while also having a sample size low enough that the human participants could identify a clear change in sound between generations.

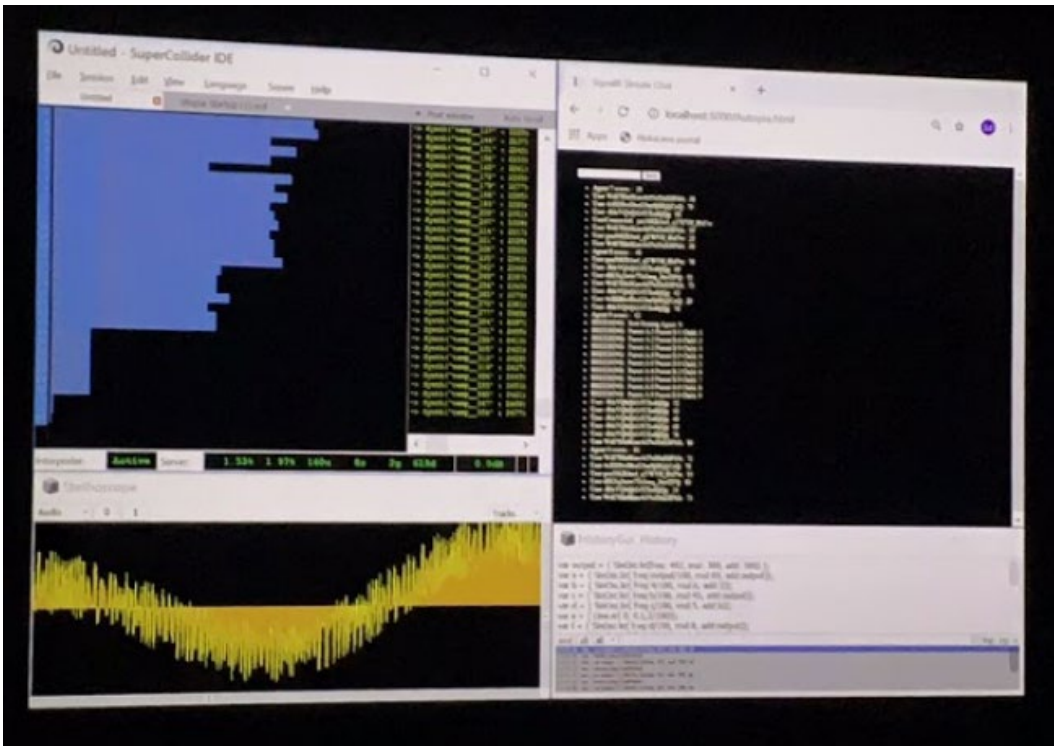


Figure 50: Photograph of Autopia.in public performance

Generated code would, rather than forming single-line solutions, form longer, complex solutions

The complete environment was labelled ‘Autopia’. This environment was utilised as part of a live-coding performance at the Academy of music and Theatre Arts, at Falmouth University. During this performance, the algorithm was left to run for an hour with direct human input from the audience. The output of the algorithm, the oscilloscope of the output signal, the Utopia interface and participants responses were projected onto a large screen (Figure 50).

The algorithm was successful in undergoing steady evolution towards various identifiable sounds that were explicitly distinguishable to the human participants as more pleasing than the initial population. The human population selected gradually changing sounds, but consistently converged around the more vocal ranges.

After the hour period, two human live coding performers joined the audio generation process, utilising Utopia’s collaborative networked interface for a further 30-minute period. This demonstrates the algorithms’ potential as an adaptive artistic medium and

as a collaborator to professional performers. Some of the sounds generated by the algorithm were adopted by the performers for their own work in other projects.

To further demonstrate this use case, a later performance was demonstrated and published [176], independently, using the same algorithm for an online audience with some success:

"*The resulting performance was in many ways a lot more immersive and engaging and received great feedback from the audience via the live stream chat. Many audience members enjoyed the novelty of an AI performer in a festival where the other performers were human, however the sounds that were generated by the AI, and particularly the sometimes-unpredictable nature of what was heard, were also received positively.*"

This experiment also direct integration of generated code into the performers' own work.

This series of experiments evidences the algorithm can evolve useful, practical solutions in human-guided environments without specialist input or even programming domain knowledge and can operate with dramatically varied template libraries and coding languages.

The code generated in these experiments was also utilised effectively by human musicians, integrating generated code into their own live-coding as part of the performance. As this implementation was in 2019, this algorithm pre-dates Larg-Language-Models for live coding for audio.

Findings from this implementation are as follows:

1. GLGPPS can generate code in a human coding pipeline.
2. GLGPPS can operate in multiple languages.
3. GLGPPS evolves effective, useful, positively received solutions to implicit fitness problems in a timeframe which meet human needs.

Chapter 6. Conclusions

This thesis introduces a new genetic algorithm from program synthesis, GLGPPS, which enforces line-by-line code scaffold construction, producing lines of finite, pre-defined length and complexity with comparative success in terms of code construction.

A simple redundant code filter is applied as part of this algorithm which demonstrates substantial improvement of generated source code. The scaffolding construction mechanism of this algorithm supports code cleaning more effectively than benchmark algorithms, with substantial improvements to human comprehension which complying more consistently with coding conventions.

Needleman-Wunsch re-alignment and Template-Based heuristic search patterns are also explored in thesis to demonstrate both compatibility and utility of the core algorithm in associated search spaces.

6.1: Contributions to knowledge

This thesis show, for the first time, that:

OneMax:

1. Needleman-Wunsch does not demonstrate notable improvement in elitest ranking searches but does increase algorithmic complexity.

Hill Climbing:

1. GLGPPS demonstrates the capability to converge into simple, syntactically correct source code with results comparable to benchmark algorithms for completion. GLGPPS demonstrates improved human comprehension over current benchmarks.

Helmuth & Spector benchmarks (against current standard G3P solutions):

1. GLGPPS produces more maintainable code:
    a. less likely to generate magic numbers
    b. less likely to breach length guidelines
    c. less subject to horizontal bloat
    d. more compatible with linters to support code cleanliness
2. GLGPPS can identify working solutions to benchmark problems in a similar number of searches to benchmark evolutionary program synthesis models.

Artificial Life:

1. GLGPPS can generate and utilise multiple functions in a single script, including generated functions from within a generated function.
2. GLGPPS can effectively evolve an Ockham's razor to subsistence solutions in implicit fitness, Artificial Life contexts.

3. GLGPPS can consistently synthesize engaging code for use with humans, demonstrating utility in creative practice with generated code being utilised as part of a human workflow.

SuperCollider:

1. GLGPPS can generate code in a human coding pipeline
2. GLGPPS can operate in multiple languages
3. GLGPPS evolves effective, useful, positively received solutions to implicit fitness problems in a timeframe which meet human needs

6.2: GLGPPS as an Auto-coder using automatically assigned fitness

As a framework for auto-coders, the most significant contribution this approach brings is the human interpretation of the source code generated, as the formatting style and explicit access to standard functionality is closer to that of the human programmer than existing methodologies in the field.

As demonstrated in the initial prototype, GLGPPS can apply hard coded statements into an automatically defined framework to generate syntactically complete, executable, standalone files following standard C-based programming syntax paradigms to create source code.

This model, at its core, follows the traditional gradient descent of genetic algorithms when applied against an explicit fitness function, successfully converging rapidly against simple hill climbing tests, though with very limited success in against unguided unit-test completions.

As this algorithm optimises its search space through an exposable genetic algorithm, this search methodology can utilise many alternative breeding methodologies seen in alternative GA frameworks: metaheuristic or re-combinatorial methods [27], [177], [178].

As this method is also generating source code freely, it only needs to be primed with decompiled code snippets. These samples may be from any reference language, the algorithm does not explicitly rely on live reflection to generate code, only to execute live in C#. A version of this algorithm could be generated in a fully interpreted language, for example, JavaScript, which does not rely on an explicit imported dynamic compiler, reflection, or delegates. Depending on the specific syntax of some languages, there may need to be some modification to the (current) interpreter.

The method also allows the likelihood for entry in the code snippet scaffold library to be modified. This allows a more heuristically optimised search space if the approximate likelihood for each entry to occur are known. For example, if it is anticipated that a terminating bracer ‘}’ may be more likely to be called than a ‘for-loop’, the appearance ratio may be set higher for the bracer.

As the evolutionary algorithm at the core of this system is modular and exposable, it can be replaced or augmented with alternative evolutionary methods, which may provide improved results in the context of a source code auto-coder.

Due to relying on genetic algorithms at its core, GLGPPS is limited by Yampolskiy's identification of evolutionary auto-coder's shortcomings [125]. It will exponentially drop off in its efficiency of exploration of its search space as it explores linearly increasing complexity solutions. This will introduce a need for heuristic approaches or explicit intervention modifying the approaches to solve long form, complex problems.

It is notable that further adoption of biologically inspired optimisation techniques may yet prove beneficial in further reducing the impact of this limitation: an argument built on a 2003 essay by Spector [179], *"Even in work that appears quite distant from biology on the surface, one often finds arguments that aspects of the proposed techniques were motivated by some hither-to under-appreciated feature of biological systems. Insofar as the adaptive power of biological systems still vastly outstrips that of any currently existing human-engineered computational system this is quite reasonable, and it is a trend that we should expect to continue."*

6.3: GLGPPS as a Creative Automatic-Programmer Using Collaborative, Human-Mediated Fitness

The main contribution of the use case of this algorithm from the SuperCollider collaborative music generation program is as a collaborative agent and as a proof of concept towards an auto-coder capable of rapid evolution with enough speed to demonstrate a noticeable change in behaviour during human interaction. GLGPPS demonstrated its ability to adapt to human input as a fitness function in real time at a rate fast enough to demonstrate clearly identifiable changes in its phenotype from a shared group experience during the performance and from an individual's training input during prototyping. This extends to demonstrating a single phenotype at a time to an audience, where the audience can recognise change in the phenotype and, as a cohort, changing the phenotype through selection in accordance with a collective preference.

As the experiment only demonstrated phenotypes which were played for ten second intervals, the algorithm would generally produce either brief fluctuating tones or very simple melodies; the experiment was run to explore human collaboration and use as a tone generator for collaboration with human musicians, rather than to analyse the algorithms' actual ability to generate music.

It is anticipated that this algorithm will again hit Yampolskiy's principle [125] in attempting to generate larger samples beyond simple repeating melodies for larger phenotype duration samples. It is possible that re-utilisation of the templating mechanism may partially address this potential limitation, though the limitation and the solution spaces have not yet been explored in this context.

One of the more significant aspects of the AI audio collaborator project was that SuperCollider was running on a completely independent, pre-compiled, Java based executable. As this operated on a text-only output, the convert-compile-commit-delegate loop was completely ignored. The output was automatically typed using simulated keypresses into the SuperCollider environment, so we can infer that:

- GLGPPS can write for multiple languages: The C# server environment successfully demonstrated the generation of SuperCollider code, demonstrating compatibility with different paradigms and programming languages, though further experimentation would allow a more robust exploration of limitations.
- GLGPPS does not require reflection to operate: GLGPPS does utilise reflection to execute natively within C#, but does not require explicit reflection calls when writing for external environments. This algorithm, when written in languages which are interpreted, should also not require reflection, though this experiment is yet to be conducted.
- GLGPPS can converge fast enough to be compatible with a non-specialist human audience when applied under suitable circumstances.
- GLGPPS can be used as an augmentation to alongside human programmers, by modifying existing code in a live coding environment, with some limitations.

To clarify why this system passes mutual goal attainment but fails at understanding and progress tracking: the objective and measurement of both human and AI agents in this system is the same, to optimise human feedback against a human audience according to quantitative feedback. Even task co-management is present as the collaboration is towards the same source code, but only the human is pre-emptive of the AI's output and the AI will generate its code independently of the human collaborator.

The lack of pre-emption, in general, is a side effect of the core basis of the algorithm: the genetic approach is inherently reactionary and may only appear to be pre-emptive. This is emphasized by the fitness function being driven by the participants: the participants will drive change in the species explicitly, with a level of pre-emptive direction, but the species can only drive change implicitly as a reflection of adaption lead by the human interactors. The human interactors may anticipate a change and steer evolution towards or away from it, while the evolutionary process itself has no explicit process of anticipation.

6.4: GLGPPS & TB-GLGPPS Under Implicit Fitness as a Controller for Artificial Life Simulations

These results are what we expect due to the balance between the mutation ratio stability and the environmental pressures the species undergoes, where species which mediate between the two pressures retain a more optimal fitness than a species which optimises into either pressure. For the Artificial Life environments explored in this model, the environmental pressures will vary over time, meaning the optimal search space is consistently moving, making convergence less likely and optimisation inconsistent in a gradualistic evolutionary model.

This results in short phenotypical solutions that specialise in genotypical stability at crossover. Due to these solutions converging to resolve their phenotype at the beginning of their gene sequence, or primarily utilise indentation and code depth as a form of redundancy for crossover stability, the geometric optimisation mechanism does not have the opportunity to impact the optimisation of search space in these agents.

The short working phenotypes of the solutions species evolved are tied to the implicit fitness function of the environment. These typically resolve the genotypically simplest solutions that are survivable in the working ecosystem. Ecological survivability and genetic functionality loss due to mutation create an environment which trends towards simple solutions, which to some extent mirrors behavioural adaptations in biological ecosystems.

Despite the short length of solutions, there are complex emergences in behaviour of the virtual species in these experiments. These produce interactions between sub-species or distinct common behavioural patterns which are clearly identifiable in all longitudinal experiments that do not undergo an early extinction event. These behaviours are usually survival strategies, such as food seeking or clustering mechanisms, though do not usually explicitly involve the location or utilisation of other agents, despite producing visually complex interactions between agents. It appears that the complexity of the evolved behaviour is dependent on the environment itself: if the environment requires multiple factors for survival, such as predator evasion or food seeking behaviours, they are more likely to occur, as these pressures increase in severity, the likelihood of them appearing increase.

The genetic systems within these agents tend to display structural genetic strategies which parallel biological genetic structures and are clearly observable across the various solutions when observing generated solution spaces. Compiled genetic structures observed demonstrate:

- Non-coding regions, which demonstrate explicit non-executable regions, likely evolved as an evolutionary strategy to mitigate detrimental mutation in longer genetic structures.
- Homology and traceable speciation as common genetic patterns remain within the species for several generations beyond initial construction of distinct phenotypically significant genes.
- Gene amplification, where phenotypically significant genes are replicated multiple times through some species genomes.
- Punctuated equilibrium, following events which trigger speciation, usually from major events which alter the survivability of the population and lead to distinct species to evolve or become dominant.

Human co-evolution in these environments occurs where a feedback cycle is produced between the evolution of the species and human interactors. The experiments have demonstrated that this can occur within a longitudinal experiment with and without prompt or when intentional pre-existing intent to interject into and modify phenotypical behaviours in a virtual species exist.

6.5: Discussion

The aim of this project was to explore co-evolution with human interactors using a novel source-code generating algorithm.

GLGPPS successfully provides a novel evolutionary mechanism which can generate more conventionally readable source code for human workflow integration, capable of evolving fast enough, in the right context, to adapt to human interaction, in real time.

This algorithm expands on existing methods for evolutionary algorithms by opening methods for internal compilation within intermediate language environments and compatibility for more complex coding structures in generated source-code, such as for-loops and if-statements. However, this algorithm ultimately holds the same limitations of all contemporary evolutionary algorithms which map programs linearly to genetic sequences, ultimately leaving this approach as optimal within contemporary automatic source code programming algorithms only for specific use cases.

The impact of institutional lockdown from Coronavirus slowed development and implementation, the direction of experimentation was also modified to comply with updated health and safety regulations, ultimately changing the direction of thesis entirely. The experiments explored in this thesis identify clear use cases of GLGPPS with repeatable, successful outcomes in human-facing, creative practice.

GLGPPS is compatible with many potential genetic operators, which may demonstrate variable speeds and ratios of completion. Due to the genetic underpinning, we see similar results to existing evolutionary algorithms for automatic code construction in terms of logarithmic performance drop off over search space size.

Against the contemporary algorithms which utilise neural networks, GLGPPS does not require large scale database and can operate with much smaller scale systems; it can prototype much faster without training but is limited to niche use-cases that are compatible with hill-climbing search-spaces due to the nature of genetic systems. In general, GLGPPS is not as effective as some contemporary algorithms for traditional programming challenges which have been developed during this thesis for complex solution spaces. Codex [180] and its implementation in GitHub Copilot [133] have been developed in this time frame, which utilise a generative pre-trained transformer from natural language representations which can operate with a higher level of abstraction than classical genetic searches. This makes Codex far more successful in most classical programming problem spaces, largely due to pre-training from human programming, at the cost of substantially larger operating cost.

Further work could explore larger scale experimentation for benchmarking or larger population co-creative practice or improve search space optimisation and complexification strategies which may mitigate some of these issues for certain use-cases and may open this algorithm to more utilitarian research.

6.6: Summary of Conclusions

GLGPPS outperforms contemporary algorithms in terms of evolutionary program synthesis algorithms for human comprehension and demonstrates comparable completion to benchmark algorithms for solving benchmark problems.

This improvement in human comprehension implies a substantial paradigm shift in the approach to program synthesis – namely, a change to the interpretation of BNF grammars to restrict line length, stacks to control variable, indentation and specific states (such as for loops).

This novel, language-agnostic program synthesis algorithm can be replicated in interpreted languages and languages with reflection for the "convert-compile-commit-delegate" loop. It can also generate source code within the host application or external applications, when provided an alternative feedback loop mechanism, such as synchronous models which demonstrate successful generation from many-interactor, single-interactor, workflow and rapid feedback models.

This algorithm can efficiently explore search spaces with a clear gradient towards a single point but becomes exponentially more expensive in linearly increasing search spaces. This is the basis of Yampolskiy's argument, and there is no evolutionary algorithm which has escaped this – it likely stems from the nature of genetically inspired algorithms themselves. Modifications could be made to the algorithm and the search space to optimise for maximum complexity against search cost, but the algorithm will inherently be limited to exponential search cost against linear problem space expansion. Notably, there is no existing solution to exponentially increasing search cost in linearly increasing problem space from any field.

The introduction of a Template-Based system demonstrates how heuristic methods can both significantly improve or hamper optimisation models without manual refinement. Both the least stable and most complex models of GLGPPS tested were Template-Based heuristic models under different conditions. The templating mechanism has shown to be successful at increasing the likelihood of the development of complex agents, though the simplest solution models consistently dominate generated populations with this algorithm.

The Needleman-Wunsch algorithm has demonstrated no significant change to elitest ranking searches, but future work may demonstrate benefits to many-agent simulations with medium to large gene length species.

6.7: Future Work

The purpose of this thesis was to introduce and explore a novel evolutionary algorithm. In doing so, this project has left a range of potential further research opportunities related to the algorithm's potential use cases, further optimisation and potential modification.

Automatic code de-compilation

To automatically construct a heuristic search space, a system which parses human-written source code could be implemented to generate a new library of code scaffolds to utilise in the auto-coder. This could be achieved by determining the functionality of a line of code and breaking down into its constituent parts, stripping out explicit variable use and setting and replacing them with the main scaffold framework.

As this system would allow the algorithm to construct a library using the target applications own code, it would be more likely to derive code fitting the format and generic architecture of the target application. This would increase the speed and general success of the algorithm within its applicable search space, though construction of this method would need heavy heuristic optimisation.

This system could be combined with an algorithm which determines, from a pool of applications, which type of problem is being solved to construct a weighting matrix for a scaffold library to aid automatic construction. This could automatically determine the context of the application to select from existing scaffolds and hierarchically organise snippets against the likelihood of being relevant.

Complete linter integration and Automatic code optimisation

For a human programmer, intelligent code completion [181] systems will automatically detect and indicate compiler warnings that are not terminatory. Applying the same automatic corrections, removal of redundant code snippets and removal of loops and trigger statements which can never trigger, dramatically reducing build-up of non-coding regions, particularly in auto-coders deriving from variable length genetic algorithms. Due to the source code generation format of this algorithm, it is directly compatible with existing, intelligent code completion systems.

A Hierarchy of Automatically Defined Functions

An expansion the Scaffold-Based approach would be to automatically apply method extraction and code-re use through refactoring, while preventing self-referential calls. This would be applied by adding a function call to the newly generated function as a new entry to the snippet scaffold library itself and therefore allowing the re-use of common functionality. Effectively, this would be the automatic formation of scaffolds, using multiple lines of a genetic sequence, into a scaffold library.

An intuitive downside to this approach is the explicit requirement to increase search space to explicitly explore random assignments of refactorizations in an already high dimensional search space.

Artificially Enforced Punctuated Equilibrium

One of the models not explored in this thesis is the use of artificial punctuated equilibrium, taking the form of a temporary inflation to mutation ratios. This would be simple to implement into the framework used in this thesis, however it was not adopted as the rate of evolution appeared to occur at a high enough rate to avoid the need for heightened mutation. The initially proposed implementation would assess modifications to the local geometry and enforce an increased mutation ratio to any breeding in a region surrounding the modification. The hypothesis being that it would enable more rapid exploration of the behavioural search space, making the species more capable of adapting to sudden ecological changes which would otherwise terminate the species.

Chapter 7. Appendix

7.1: Psuedocode

Figure 51: 2D Needleman-Wunsch algorithm

```
1.  Input: 2D integer arrays seq1, seq2; target as a 2D integer array
2.  Output: Aligned target
3.
4.  // Initialization
5.  R ← new Random
6.  linesMin ← min(|seq1|, |seq2|), linesMax ← max(|seq1|, |seq2|)
7.  seq1Larger ← (|seq1| > |seq2|)
8.  preExistingContent ← range(1, linesMin); nSpaces ← linesMax
9.  theOrder ← ""; GenerateArrangements(preExistingContent, nSpaces, [], 0,
        theOrder)
10. arrangements ← split(theOrder, ".")
11. bestScore ← − ∞, bestID ← −1
12.
13. // Process each arrangement
14. for each arrangement in arrangements do
15.   if arrangement ≠ "" then
16.     myData ← parse(arrangement)  // list of ints
17.     scoreTotal ← 0; currentFunction ← []
18.     for k from 0 to linesMin − 1 do
19.       if myData[k] ≠ −1 then
20.         if seq1Larger then
21.           n ← length(seq1[k]); m ← length(seq2[myData[k] − 1])
22.         else
23.           n ← length(seq1[myData[k] − 1]); m ← length(seq2[k])
24.         Set matchScore = 1, mismatch = −1, gap = −1
25.         Initialize scoreMatrix[0..n, 0..m]:
26.           scoreMatrix[i,0] = i * gap, scoreMatrix[0,j] = j * gap
27.         for i = 1 to n do
28.           for j = 1 to m do
29.             match ← scoreMatrix[i−1,j−1] + (equal ? matchScore :
                    mismatch)
30.             delete ← scoreMatrix[i−1,j] + gap; insert ← scoreMatrix[i,j−1]
                    + gap
31.             scoreMatrix[i,j] ← max(match, delete, insert)
32.         (x,y) ← (n, m); currentLine ← []
```

```
33.            While (x > 0 or y > 0) do
34.              if (x > 0 and scoreMatrix[x,y] = scoreMatrix[x−1,y] + gap) then
35.                 with probability 0.5: prepend element from
                                seq1 (or seq1[myData[k]−1]) to currentLine; x ← x − 1
36.              else if (y > 0 and scoreMatrix[x,y] = scoreMatrix[x,y−1] + gap)
                          then
37.                 with probability 0.5: prepend element from seq2 (or
                                seq2[myData[k]−1]) to currentLine; y ← y − 1
38.              else
39.                 prepend element from seq1 (or seq1[myData[k]−1]) else from
                                seq2; x ← x − 1; y ← y − 1
40.              scoreTotal ← scoreTotal + scoreMatrix[n, m]
41.              Append deep copy(currentLine) to currentFunction
42.           else if (R.NextDouble() ≥ 0.5 or currentFunction is empty) then
43.              Append deep copy(currentLine) to currentFunction
44.           if scoreTotal > bestScore then
45.                 bestScore ← scoreTotal; bestID ← current ID; target ←
                                deep copy(currentFunction)
46.        Reset currentFunction and scoreTotal
47. end for
48. // Post-alignment adjustment
49. for j from 0 to 4 do
50.    for i from 0 to |target| − 1 do
51.       if j not in bounds of target[i] then
52.          if seq1[i][j] and seq2[i][j] exist then
53.             with probability 0.5: insert seq1[i][j] at position j in target[i]
                          else insert seq2[i][j]
54.          else if seq1[i][j] exists then insert seq1[i][j]
55.          else if seq2[i][j] exists then insert seq2[i][j]
56. return target
```

Figure 52: GLGPPS Allele to code mapping algorithm

```
1.  Input:
2.          GeneticSequence: list of lists of integers (genetic code)
3.          outputCode: string (by reference) — the script to append generated code
4.          indentDepth: integer (by reference) — current indentation level
5.          valuesAnnounced: integer (by reference) — count of announced values
6.          i: integer — index for the current genetic sequence
7.          lastAdded: integer
8.          valueList: list of integers (by reference)
9.          data: list of lists of strings — code scaffold templates
10.         stack: list of integers  — variable scope stack
11.
12. Output:
13.         Updated outputCode and indentDepth reflecting the appended code
14.
15. // Preserve the scaffold value to avoid modification
16. scaffoldValue ← DeepCopy(GeneticSequence[i][0])
17. if (GeneticSequence[i][0] + 1 > Count(data)) then
18.         return
19.
20. codeLength ← Count(data[GeneticSequence[i][0]])
21. codonScaffold ← 0
22. codonGA ← 1
23. firstInput ←ValueFormatter( data[ GeneticSequence[i][0]][0],
            GeneticSequence[i][1],  valuesAnnounced, lastAdded, valueList, 0)
24.
25. // Adjust indentation for loop constructs
26. if (data[GeneticSequence[i][0]][0] equals "for (double ") then
27.         indentDepth ← indentDepth + 1
28.         Call VariableScopeController(indentDepth, valuesAnnounced, stack,
                valueList)
29.
30. // Handle closing braces from the scaffold
31. if (data[GeneticSequence[i][0]][0] equals "}") then
32.         if (indentDepth < 1) then
33.                 return
34.         end if
35.         indentDepth ← indentDepth − 1
36.         outputCode  ← outputCode + Indent(indentDepth + 4) + "}"
37.         return
38. end if
39.
40. if (FirstCharacter(data[GeneticSequence[i][0]][0]) equals '}') then
41.         if (indentDepth < 1) then
42.                 return
43.         end if
```

```
44.         indentDepth ← indentDepth − 1
45.         Call VariableScopeController(indentDepth, valuesAnnounced, stack,
                valueList)
46.         indentDepth ← indentDepth + 1
47.         outputCode  ← outputCode + Indent(indentDepth + 3)
48. else
49.         outputCode  ← outputCode + Indent(indentDepth + 4)
50. end if
51.
52. // Append the first scaffold input if it exists
53. if (firstInput is not empty) then
54.         outputCode  ← outputCode + firstInput
55.         codonGA             ← codonGA + 1
56.         codonScaffold ← codonScaffold + 1
57. end if
58.
59. swap ← TRUE
60. while (codonScaffold < codeLength - 1) do
61.         if (swap equals TRUE) then
62.                 outputCode  ← outputCode +
    data[GeneticSequence[i][0]][codonScaffold]
63.                 codonScaffold ← codonScaffold + 1
64.         else
65.                 outputCode  ← outputCode +
                        ValueFormatter(data[GeneticSequence[i][0]][codonScaf
                        fold], GeneticSequence[i][codonGA], valuesAnnounced,
                        lastAdded, valueList, codonScaffold)
66.                 codonScaffold ← codonScaffold + 1
67.                 codonGA             ← codonGA + 1
68.         end if
69.         swap ← NOT swap
70. end while
71.
72. depthHolder ← DepthSetter(data[GeneticSequence[i][0]][codeLength - 1])
73. if (indentDepth > 0 or depthHolder > 0) then
74.         indentDepth ← indentDepth + depthHolder
75. end if
76.
77. // Restore original scaffold value to maintain template integrity
78. GeneticSequence[i][0] ← scaffoldValue
```

7.2: Helmuth and Spector Benchmark Graphs

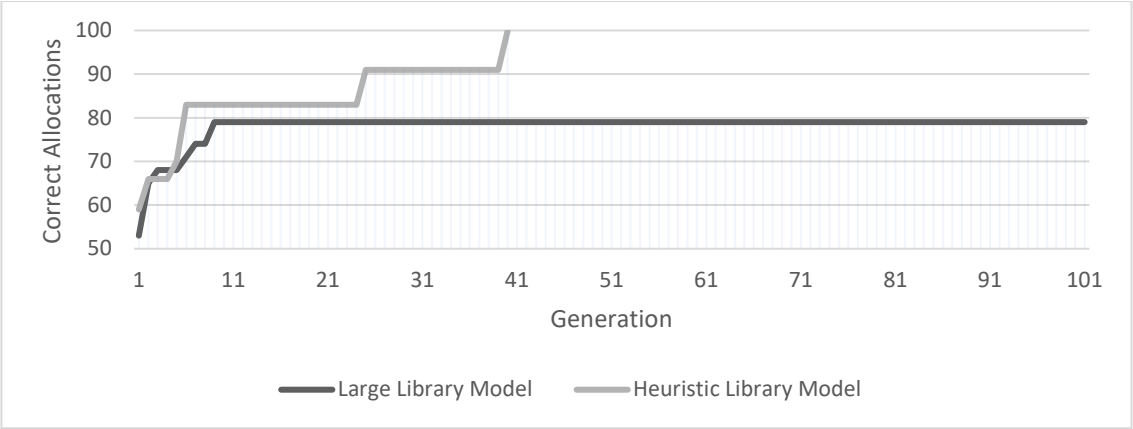


Figure 53: Graph of GLGPPS first solution for "MEDIAN" Benchmark test

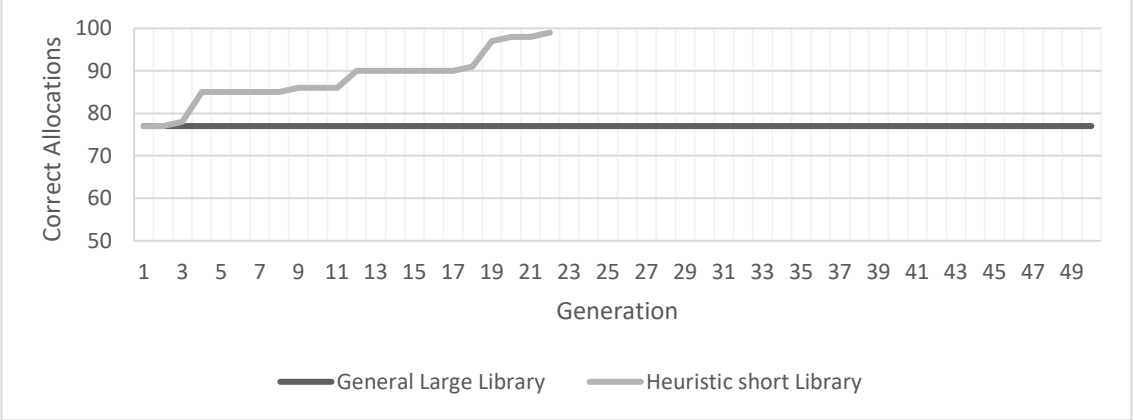


Figure 54: Graph of GLGPPS first solution for "SMALL OR LARGE" Benchmark test

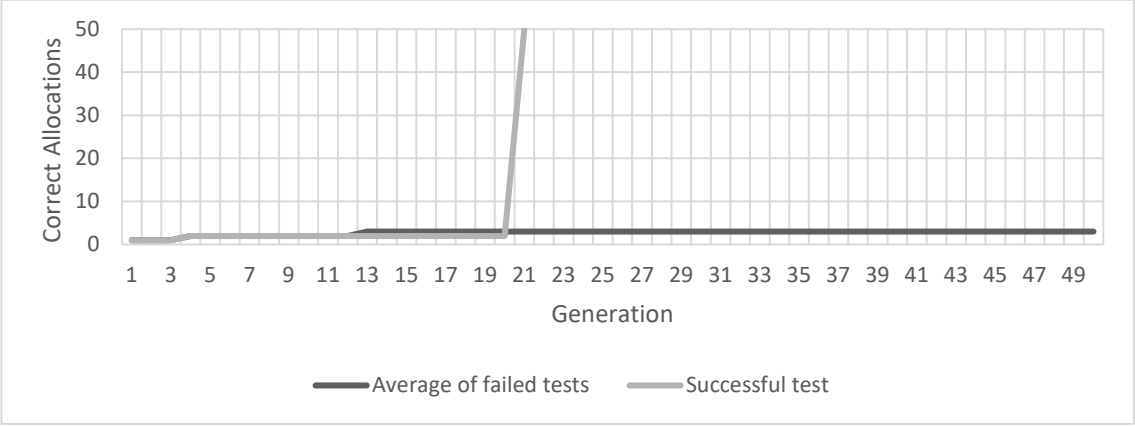


Figure 55: Graph of GLGPPS first solution for "SUM OF SQUARES" Benchmark test

Chapter 8. References

[1] D. Sobania, J. Petke, M. Briesch, and F. Rothlauf, “A Comparison of Large Language Models and Genetic Programming for Program Synthesis,” *IEEE Trans. Evol. Comput.*, pp. 1–1, 2024, doi: 10.1109/TEVC.2024.3410873.

[2] L. Spector, “Autoconstructive Evolution: Push, PushGP, and Pushpop,” *GECCO-2001 Proc. Genet. Evol. Comput. Confer-Ence*, 2001.

[3] “Copilot+ PC hardware requirements - Microsoft Support.” Accessed: Jul. 19, 2024. [Online]. Available: https://support.microsoft.com/en-gb/topic/copilot-pc-hardware-requirements-35782169-6eab-4d63-a5c5-c498c3037364

[4] G. R. Harik, F. G. Lobo, and D. E. Goldberg, “The compact genetic algorithm,” *IEEE Trans. Evol. Comput.*, vol. 3, no. 4, pp. 287–297, Nov. 1999, doi: 10.1109/4235.797971.

[5] J. Austin *et al.*, “Program Synthesis with Large Language Models,” Aug. 15, 2021, *arXiv*: arXiv:2108.07732. Accessed: Jul. 19, 2024. [Online]. Available: http://arxiv.org/abs/2108.07732

[6] P. Naur *et al.*, “Revised Report on the Algorithmic Language Algol 60,” p. 43, 1962.

[7] T. Helmuth and L. Spector, “General Program Synthesis Benchmark Suite,” in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, in GECCO ’15. New York, NY, USA: Association for Computing Machinery, Jul. 2015, pp. 1039–1046. doi: 10.1145/2739480.2754769.

[8] W. Crichton, “Human-Centric Program Synthesis,” in *OASIcs, Volume 76, PLATEAU 2019*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020, p. 5:1-5:5. doi: 10.4230/OASICS.PLATEAU.2019.5.

[9] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, “Measuring program comprehension: a large-scale field study with professionals,” *Proc. 40th Int. Conf. Softw. Eng.*, pp. 584–584, May 2018, doi: 10.1145/3180155.3182538.

[10] R. Minelli, A. Mocci, and M. Lanza, “I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time,” in *2015 IEEE 23rd International Conference on Program Comprehension*, May 2015, pp. 25–35. doi: 10.1109/ICPC.2015.12.

[11] D. R. Wallace, A. H. Watson, and T. J. McCabe, “Structured testing : a testing methodology using the cyclomatic complexity metric,” National Institute of Standards and Technology, Gaithersburg, MD, NIST SP 500-235, 1996. doi: 10.6028/NIST.SP.500-235.

[12] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. USA: Elsevier Science Inc., 1977.

[13] P. Oman and J. Hagemeister, “Construction and testing of polynomials predicting software maintainability,” *J. Syst. Softw.*, vol. 24, no. 3, pp. 251–266, Mar. 1994, doi: 10.1016/0164-1212(94)90067-1.

[14] Mikejo5000, “Code metrics - Maintainability index range and meaning - Visual Studio (Windows).” Accessed: Jul. 21, 2024. [Online]. Available: https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=vs-2022

[15] N. Verma, S. Manvati, and P. Dhar, “Chapter 15 - FLAGSHIP: A novel drug discovery platform originating from the ‘dark matter of the genome,’” in *Translational Biotechnology*, Y. Hasija, Ed., Academic Press, 2021, pp. 371–379. doi: 10.1016/B978-0-12-821972-0.00011-3.

[16] G. Litwack, “Chapter 10 - Nucleic Acids and Molecular Genetics,” in *Human Biochemistry*, G. Litwack, Ed., Boston: Academic Press, 2018, pp. 257–317. doi: 10.1016/B978-0-12-383864-3.00010-7.
[17] A. S. Wu, R. K. Lindsay, and M. D. Smith, “Studies on the effect of non-coding segments on the genetic algorithm,” in *Proceedings Sixth International Conference on Tools with Artificial Intelligence. TAI 94*, Nov. 1994, pp. 744–747. doi: 10.1109/TAI.1994.346411.
[18] A. S. Wu and R. K. Lindsay, “Empirical Studies of the Genetic Algorithm with Noncoding Segments,” *Evol. Comput.*, vol. 3, no. 2, pp. 121–147, Jun. 1995, doi: 10.1162/evco.1995.3.2.121.
[19] “Genetic Algorithms - John H. Holland.” Accessed: Dec. 04, 2019. [Online]. Available: http://www2.econ.iastate.edu/tesfatsi/holland.GAIntro.htm
[20] W. Banzhaf and W. B. Langdon, “Some Considerations on the Reason for Bloat,” *Genet. Program. Evolvable Mach.*, vol. 3, no. 1, pp. 81–91, Mar. 2002, doi: 10.1023/A:1014548204452.
[21] T. Blickle, L. Thiele, and L. F. Mikroelektronik, “Genetic Programming and Redundancy.” 1994.
[22] C. Biémont and C. Vieira, “Junk DNA as an evolutionary force,” *Nature*, vol. 443, no. 7111, pp. 521–524, Oct. 2006, doi: 10.1038/443521a.
[23] J. O. Andersson and S. G. E. Andersson, “Pseudogenes, Junk DNA, and the Dynamics of Rickettsia Genomes,” *Mol. Biol. Evol.*, vol. 18, no. 5, pp. 829–839, May 2001, doi: 10.1093/oxfordjournals.molbev.a003864.
[24] A. L. Panchen, “Homology--history of a concept,” *Novartis Found. Symp.*, vol. 222, pp. 5–18; discussion 18-23, 1999, doi: 10.1002/9780470515655.ch2.
[25] L. Gabora, “Convergent Evolution,” in *Brenner’s Encyclopedia of Genetics (Second Edition)*, S. Maloy and K. Hughes, Eds., San Diego: Academic Press, 2013, pp. 178–180. doi: 10.1016/B978-0-12-374984-0.00336-3.
[26] G. Rudolph, “Convergence analysis of canonical genetic algorithms,” *IEEE Trans. Neural Netw.*, vol. 5, no. 1, pp. 96–101, Jan. 1994, doi: 10.1109/72.265964.
[27] C. Headleand, “Grammatical Herding,” *J. Comput. Sci. Syst. Biol.*, vol. 06, no. 02, 2013, doi: 10.4172/jcsb.1000099.
[28] O. Honnay, “Genetic Drift,” in *Brenner’s Encyclopedia of Genetics (Second Edition)*, S. Maloy and K. Hughes, Eds., San Diego: Academic Press, 2013, pp. 251–253. doi: 10.1016/B978-0-12-374984-0.00616-1.
[29] G. Thomson, “Gametic Disequilibrium,” in *Encyclopedia of Genetics*, S. Brenner and J. H. Miller, Eds., New York: Academic Press, 2001, pp. 750–752. doi: 10.1006/rwgn.2001.0493.
[30] D. Graur and W.-H. Li, *Fundamentals of Molecular Evolution*. Sinauer, 2000.
[31] J. W. Szostak and R. Wu, “Unequal crossing over in the ribosomal DNA of Saccharomyces cerevisiae,” *Nature*, vol. 284, no. 5755, pp. 426–430, Apr. 1980, doi: 10.1038/284426a0.
[32] J. E. Darnell and W. F. Doolittle, “Speculations on the early course of evolution.,” *Proc. Natl. Acad. Sci.*, vol. 83, no. 5, pp. 1271–1275, Mar. 1986, doi: 10.1073/pnas.83.5.1271.
[33] A. Force, M. Lynch, F. B. Pickett, A. Amores, Y. L. Yan, and J. Postlethwait, “Preservation of duplicate genes by complementary, degenerative mutations.,” *Genetics*, vol. 151, no. 4, pp. 1531–1545, Apr. 1999.
[34] A. G. Uren, J. Kool, A. Berns, and M. van Lohuizen, “Retroviral insertional mutagenesis: past, present and future,” *Oncogene*, vol. 24, no. 52, pp. 7656–7672, Nov. 2005, doi: 10.1038/sj.onc.1209043.

[35] D. Leister and T. Kleine, “Chapter three - Role of Intercompartmental DNA Transfer in Producing Genetic Diversity,” in *International Review of Cell and Molecular Biology*, vol. 291, K. W. Jeon, Ed., Academic Press, 2011, pp. 73–114. doi: 10.1016/B978-0-12-386035-4.00003-3.
[36] S. J. Gould and N. Eldredge, “Punctuated equilibria: the tempo and mode of evolution reconsidered,” *Paleobiology*, vol. 3, no. 02, pp. 115–151, 1977, doi: 10.1017/S0094837300005224.
[37] Z. Ma, G. G. Turrigiano, R. Wessel, and K. B. Hengen, “Cortical Circuit Dynamics Are Homeostatically Tuned to Criticality In Vivo,” *Neuron*, vol. 104, no. 4, pp. 655-664.e4, Nov. 2019, doi: 10.1016/j.neuron.2019.08.031.
[38] K. C. Falke, S. Glander, F. He, J. Hu, J. de Meaux, and G. Schmitz, “The spectrum of mutations controlling complex traits and the genetics of fitness in plants,” *Curr. Opin. Genet. Dev.*, vol. 23, no. 6, pp. 665–671, Dec. 2013, doi: 10.1016/j.gde.2013.10.006.
[39] R. Jonnal and A. Chemero, “Punctuation Equilibrium and Optimization: An A-Life Model,” p. 8.
[40] J. F. Nash, “Equilibrium points in n-person games,” *Proc. Natl. Acad. Sci.*, vol. 36, no. 1, pp. 48–49, Jan. 1950, doi: 10.1073/pnas.36.1.48.
[41] J. Nash, “Non-Cooperative Games,” *Ann. Math.*, vol. 54, no. 2, pp. 286–295, 1951, doi: 10.2307/1969529.
[42] S. A. Kaufmann and S. Johnsen, “Coevolution to the Edge of Chaos: Coupled Fitness Landscapes, Poised States, and Coevolutionary Avalanches,” *J. Theor. Biol.*, Apr. 1991, doi: 10.1016/S0022-5193(05)80094-3.
[43] K. A. Peacock, “Symbiosis in Ecology and Evolution,” in *Philosophy of Ecology*, vol. 11, K. deLaplante, B. Brown, and K. A. Peacock, Eds., in Handbook of the Philosophy of Science, vol. 11. , Amsterdam: North-Holland, 2011, pp. 219–250. doi: 10.1016/B978-0-444-51673-2.50009-1.
[44] J. N. Thompson, “Four Central Points About Coevolution,” *Evol. Educ. Outreach*, vol. 3, no. 1, Art. no. 1, Mar. 2010, doi: 10.1007/s12052-009-0200-x.
[45] N. Bacaër, “Lotka, Volterra and the predator–prey system (1920–1926),” *Short Hist. Math. Popul. Dyn.*, pp. 71–76, 2011, doi: 10.1007/978-0-85729-115-8_13.
[46] Y. Takeuchi, N. H. Du, N. T. Hieu, and K. Sato, “Evolution of predator–prey systems described by a Lotka–Volterra equation under random environment,” *J. Math. Anal. Appl.*, vol. 323, no. 2, pp. 938–957, Nov. 2006, doi: 10.1016/j.jmaa.2005.11.009.
[47] T. C. Morris and M. J. Costello, “The Biology, Ecology and Societal Importance of Marine Parasites,” in *Encyclopedia of the World’s Biomes*, M. I. Goldstein and D. A. DellaSala, Eds., Oxford: Elsevier, 2020, pp. 556–566. doi: 10.1016/B978-0-12-409548-9.11802-0.
[48] E. G. LEIGH Jr, “The evolution of mutualism,” *J. Evol. Biol.*, vol. 23, no. 12, pp. 2507–2528, 2010, doi: 10.1111/j.1420-9101.2010.02114.x.
[49] D. W. Mount, *Bioinformatics: Sequence and Genome Analysis*. CSHL Press, 2004.
[50] S. B. Needleman and C. D. Wunsch, “A general method applicable to the search for similarities in the amino acid sequence of two proteins,” *J. Mol. Biol.*, vol. 48, no. 3, pp. 443–453, Mar. 1970, doi: 10.1016/0022-2836(70)90057-4.
[51] W. J. Masek and M. S. Paterson, “A faster algorithm computing string edit distances,” *J. Comput. Syst. Sci.*, vol. 20, no. 1, pp. 18–31, Feb. 1980, doi: 10.1016/0022-0000(80)90002-1.
[52] L. J. Fogel, *Intelligence Through Simulated Evolution: Forty Years of Evolutionary Programming*. New York, NY, USA: John Wiley & Sons, Inc., 1999.

[53] D. B. Fogel, *Evolutionary Computation: The Fossil Record*, 1st ed. Wiley-IEEE Press, 1998.
[54] N. A. Barricelli, “Esempi Numerici di processi dievoluzione,” *Methodos*, pp. 45–68, 1954.
[55] N. A. Barricelli, “symbiogenetic evolution processes realized by artificial methods,” *Methodos*, vol. 9, no. 35–36, pp. 143–182, 1957.
[56] D. B. Fogel, “Nils Barricelli - artificial life, coevolution, self-adaptation,” *IEEE Comput. Intell. Mag.*, vol. 1, no. 1, pp. 41–45, Feb. 2006, doi: 10.1109/MCI.2006.1597062.
[57] J. Von Neumann and A. W. (Arthur W. Burks, *Theory of self-reproducing automata*. Urbana, University of Illinois Press, 1966. Accessed: Dec. 29, 2019. [Online]. Available: http://archive.org/details/theoryofselfrepr00vonn_0
[58] M. Gardner, “MATHEMATICAL GAMES,” *Sci. Am.*, vol. 223, no. 4, pp. 120–123, 1970.
[59] R. Ablowitz, “The Theory of Emergence,” *Philos. Sci.*, vol. 6, no. 1, pp. 1–16, 1939.
[60] J. G. Lennox, “Aristotle on the Emergence of Material Complexity: Meteorology IV and Aristotle’s Biology,” *HOPOS J. Int. Soc. Hist. Philos. Sci.*, vol. 4, no. 2, pp. 272–305, Sep. 2014, doi: 10.1086/677568.
[61] J. H. Holland, *Hidden order: how adaptation builds complexity*. USA: Addison Wesley Longman Publishing Co., Inc., 1996.
[62] J. H. Holland, *Emergence: From Chaos to Order*. Perseus Publishing, 1999.
[63] A. S. Fraser, “Simulation of Genetic Systems by Automatic Digital Computers I. Introduction,” *Aust. J. Biol. Sci.*, vol. 10, no. 4, pp. 484–491, 1957, doi: 10.1071/bi9570484.
[64] D. B. Fogel, “In memoriam Alex S. Fraser [1923-2002],” *IEEE Trans. Evol. Comput.*, vol. 6, no. 5, pp. 429–430, Oct. 2002, doi: 10.1109/TEVC.2002.805212.
[65] N. Metropolis and S. Ulam, “The Monte Carlo Method,” *J. Am. Stat. Assoc.*, vol. 44, no. 247, pp. 335–341, Sep. 1949, doi: 10.1080/01621459.1949.10483310.
[66] A. S. Fraser, “Monte Carlo Analyses of Genetic Models,” *Nature*, vol. 181, no. 4603, pp. 208–209, Jan. 1958, doi: 10.1038/181208a0.
[67] D. B. Fogel, “Unearthing a Fossil from the History of Evolutionary Computation,” *Fundam. Informaticae*, vol. 35, no. 1–4, pp. 1–16, 1998.
[68] R. M. Friedberg, “A Learning Machine: Part I,” *IBM J. Res. Dev.*, vol. 2, no. 1, pp. 2–13, Jan. 1958, doi: 10.1147/rd.21.0002.
[69] M. Minsky, “Steps toward Artificial Intelligence,” *Proc. IRE*, vol. 49, no. 1, pp. 8–30, Jan. 1961, doi: 10.1109/JRPROC.1961.287775.
[70] D. B. Fogel, *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. John Wiley & Sons, 2006.
[71] R. M. Friedberg, B. Dunham, and J. H. North, “A Learning Machine: Part II,” *IBM J. Res. Dev.*, vol. 3, no. 3, pp. 282–287, Jul. 1959, doi: 10.1147/rd.33.0282.
[72] H. J. Bremermann, *The evolution of intelligence: The nervous system as a model of its environment*. University of Washington, Department of Mathematics, 1958.
[73] H. J. Bremermann, “Optimization Through Evolution and Recombination,” *Self-Organ. Syst.*, no. 5, p. 12, 1962.
[74] D. B. Fogel and R. W. Anderson, “Revisiting Bremermann’s genetic algorithm. I. Simultaneous mutation of all parameters,” in *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No.00TH8512)*, Jul. 2000, pp. 1204–1209 vol.2. doi: 10.1109/CEC.2000.870787.

[75] L. J. Fogel, A. J. Owens, and M. J. Walsh, *Artificial Intelligence through Simulated Evolution*. Wiley, 1966.
[76] J. H. Holland, "Outline for a Logical Theory of Adaptive Systems," *J. ACM*, vol. 9, no. 3, pp. 297–314, Jul. 1962, doi: 10.1145/321127.321128.
[77] J. H. Holland, P. of P. and of E. E. and C. S. J. H. Holland, and S. L. in H. R. M. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992.
[78] P. F. Stadler, "Fitness Landscapes," *Appl Math Comput*, vol. 117, pp. 187–207, 2002.
[79] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. in Complex adaptive systems. Cambridge, Mass: MIT Press, 1992.
[80] R. Poli, W. Langdon, N. Mcphee, and J. Koza, "Genetic programming: An introductory tutorial and a survey of techniques and applications," Nov. 2007.
[81] R. Forsyth, "BEAGLE: A Darwinian Approach to PatternRecognition," *Kybernetes*, vol. 10, no. 3, pp. 159–166, Mar. 1981, doi: 10.1108/eb005587.
[82] R. Forsyth, "The evolution of BEAGLE: confessions of a mongrel rule-breeder," *ACM SIGEVOlution*, vol. 9, no. 1, p. 4, Aug. 2016, doi: 10.1145/2983381.2983382.
[83] A. Teller, "Turing completeness in the language of genetic programming with indexed memory," in *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, Jun. 1994, pp. 136–141 vol.1. doi: 10.1109/ICEC.1994.350027.
[84] Nichael Lynn Cramer, "A representation for the Adaptive Generation of Simple Sequential Programs," *Proc. Int. Conf. Genet. Algorithms Appl.*, 1985, Accessed: Jan. 17, 2020. [Online]. Available: http://gpbib.cs.ucl.ac.uk/gp-html/icga85_cramer.html
[85] J. R. Koza, "Non-linear genetic algorithms for solving problems by finding a fit composition of functions," US5136686A, Aug. 04, 1992 Accessed: Jan. 17, 2020. [Online]. Available: https://patents.google.com/patent/US5136686A/en
[86] J. R. Koza, *Genetic programming II: automatic discovery of reusable programs*. Cambridge, MA, USA: MIT Press, 1994.
[87] J. R. Koza, D. Andre, F. H. Bennett, and M. A. Keane, *Genetic Programming III: Darwinian Invention & Problem Solving*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.
[88] J. R. Koza, *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. USA: Kluwer Academic Publishers, 2003.
[89] J. R. Koza and J. P. Rice, "Non-linear genetic process for data encoding and for solving problems using automatically defined functions," 5343554, Aug. 30, 1994 Accessed: Jan. 17, 2020. [Online]. Available: http://www.freepatentsonline.com/5343554.html
[90] J. R. Koza, "Hierarchical Automatic Function Definition in Genetic Programming," in *Foundations of Genetic Algorithms*, vol. 2, L. D. Whitley, Ed., in Foundations of Genetic Algorithms, vol. 2. , Elsevier, 1993, pp. 297–318. doi: 10.1016/B978-0-08-094832-4.50024-6.
[91] W. S. Bruce, "Automatic Generation of Object-oriented Programs Using Genetic Programming," in *Proceedings of the 1st Annual Conference on Genetic Programming*, Cambridge, MA, USA: MIT Press, 1996, pp. 267–272. Accessed: Feb. 19, 2019. [Online]. Available: http://dl.acm.org/citation.cfm?id=1595536.1595571

[92] Wilker Shane Bruce, “The Application of Genetic Programming to the Automatic Generation of Object-Oriented Programs,” School of Computer and Information Sciences, Nova Southeastern University, 1995.
[93] W. B. Langdon, “Evolving Data Structures with Genetic Programming,” in *Proceedings of the Sixth International Conference on Genetic Algorithms*, Morgan Kaufmann, 1995, pp. 295–302.
[94] K. Nygaard and O.-J. Dahl, “The development of the SIMULA languages,” in *History of programming languages*, New York, NY, USA: Association for Computing Machinery, 1978, pp. 439–480. Accessed: Jan. 22, 2020. [Online]. Available: http://doi.org/10.1145/800025.1198392
[95] B. Meyer, *Object-Oriented Software Construction*, 1st ed. USA: Prentice-Hall, Inc., 1988.
[96] A. Agapitos and S. M. Lucas, “Evolving a Statistics Class Using Object Oriented Evolutionary Programming,” in *Genetic Programming*, M. Ebner, M. O’Neill, A. Ekárt, L. Vanneschi, and A. I. Esparcia-Alcázar, Eds., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 291–300. doi: 10.1007/978-3-540-71605-1_27.
[97] A. Agapitos and S. M. Lucas, “Evolving a Statistics Class Using Object Oriented Evolutionary Programming,” in *EuroGP*, 2007. doi: 10.1007/978-3-540-71605-1_27.
[98] N. Paterson and M. Livesey, “Evolving caching algorithms in C by genetic programming,” *Proc. 2nd Annu. Conf. Genet. Program.*, pp. 262–267, 1997.
[99] N. R. Paterson and M. Livesey, “Distinguishing genotype and phenotype in genetic programming,” *Proc. 1st Annu. Conf. Genet. Program.*, pp. 141–150, 1996.
[100] L. Spector, J. Klein, and M. Keijzer, “The Push3 execution stack and the evolution of control,” in *Proceedings of the 2005 conference on Genetic and evolutionary computation - GECCO ’05*, Washington DC, USA: ACM Press, 2005, p. 1689. doi: 10.1145/1068009.1068292.
[101] A. Agapitos and S. M. Lucas, “Learning Recursive Functions with Object Oriented Genetic Programming,” in *Genetic Programming*, P. Collet, M. Tomassini, M. Ebner, S. Gustafson, and A. Ekárt, Eds., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 166–177. doi: 10.1007/11729976_15.
[102] T. White, J. Fan, and F. Oppacher, “Basic Object Oriented Genetic Programming,” Jun. 2011, pp. 59–68. doi: 10.1007/978-3-642-21822-4_7.
[103] M. Oltean, “Liquid State Genetic Programming,” Apr. 2007, pp. 220–229. doi: 10.1007/978-3-540-71618-1_25.
[104] C. Ryan, J. Collins, and M. O. Neill, “Grammatical evolution: Evolving programs for an arbitrary language,” in *Genetic Programming*, vol. 1391, W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 83–96. doi: 10.1007/BFb0055930.
[105] M. O’Neill and C. Ryan, *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. in Genetic Programming. Springer US, 2003. doi: 10.1007/978-1-4615-0447-4.
[106] M. O’Neill, C. Ryan, and M. Nicolau, “Grammar Defined Introns: An Investigation Into Grammars, Introns, and Bias in Grammatical Evolution,” presented at the GECCO 2001 - Genetic and Evolutionary Computation Conference, San Francisco, California, USA, 7-11 July 2001, Morgan Kauffman, Jul. 2001. Accessed: Jan. 24, 2020. [Online]. Available: https://researchrepository.ucd.ie/handle/10197/8370

[107] A. Brabazon and M. O'Neill, *Biologically Inspired Algorithms for Financial Modelling*. in Natural Computing Series. Berlin Heidelberg: Springer-Verlag, 2006. doi: 10.1007/3-540-31307-9.
[108] I. Dempsey, M. O'Neill, and A. Brabazon, *Foundations in Grammatical Evolution for Dynamic Environments*. in Studies in Computational Intelligence. Berlin Heidelberg: Springer-Verlag, 2009. doi: 10.1007/978-3-642-00314-1.
[109] L. Georgiou, "Constituent Grammatical Evolution," University of Wales, Bangor, 2012.
[110] L. Georgiou and W. J. Teahan, "Constituent Grammatical Evolution," in *IJCAI*, 2011. doi: 10.5591/978-1-57735-516-8/IJCAI11-214.
[111] A. Moraglio, J. McDermott, and M. O'Neill, "Geometric Semantic Grammatical Evolution," *Handb. Gramm. Evol.*, pp. 163–188.
[112] T. Castle and C. G. Johnson, "Positional Effect of Crossover and Mutation in Grammatical Evolution," in *Genetic Programming*, A. I. Esparcia-Alcázar, A. Ekárt, S. Silva, S. Dignum, and A. Ş. Uyar, Eds., in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 26–37.
[113] H. Svensson and T. Ziemke, *Making sense of embodiment: simulation theories and the sharing of neural circuitry between sensorimotor and cognitive processes,"in Presented at the 26th Annual Cognitive Science Society Conference,Chicago,IL*. 2004.
[114] H. D. Jaegher and M. Rohde, *Enaction: Toward a New Paradigm for Cognitive Science*. MIT Press, 2010.
[115] P. de Loor, K. Manac'H, and J. Tisseau, "Enaction-Based Artificial Intelligence: Toward Co-evolution with Humans in the Loop," *Minds Mach.*, vol. 19, pp. 319–343, Oct. 2009, doi: 10.1007/s11023-009-9165-3.
[116] R. Brooks, "A robust layered control system for a mobile robot," *IEEE J. Robot. Autom.*, vol. 2, no. 1, pp. 14–23, Mar. 1986, doi: 10.1109/JRA.1986.1087032.
[117] R. A. Brooks, "Planning is Just a Way of Avoiding Figuring Out What To Do Next," MIT Artificial Intelligence Laboratory, Working Paper, Sep. 1987. Accessed: Jan. 27, 2020. [Online]. Available: https://dspace.mit.edu/handle/1721.1/41202
[118] C. J. Headleand, G. Henshall, L. A. Cenydd, and W. J. Teahan, "Towards Real-Time Behavioral Evolution in Video Games," in *Artificial Life and Intelligent Agents*, C. J. Headleand, W. J. Teahan, and L. Ap Cenydd, Eds., in Communications in Computer and Information Science. Cham: Springer International Publishing, 2015, pp. 3–16. doi: 10.1007/978-3-319-18084-7_1.
[119] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, Nov. 1995, pp. 1942–1948 vol.4. doi: 10.1109/ICNN.1995.488968.
[120] M. O'Neill and A. Brabazon, "Grammatical Swarm," in *Genetic and Evolutionary Computation – GECCO 2004*, K. Deb, Ed., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 163–174. doi: 10.1007/978-3-540-24854-5_15.
[121] M. O'Neill, F. Leahy, and A. Brabazon, "Grammatical Swarm: A Variable-Length Particle Swarm Algorithm," in *Swarm Intelligent Systems*, N. Nedjah and L. de M. Mourelle, Eds., in Studies in Computational Intelligence. , Berlin, Heidelberg: Springer, 2006, pp. 59–74. doi: 10.1007/978-3-540-33869-7_3.
[122] C. Headleand, L. Cenydd, and W. Teahan, "Berry Eaters: Learning Color Concepts with Template Based Evolution," in *Artificial Life 14: Proceedings of the Fourteenth International Conference on the Synthesis and Simulation of Living*

Systems, Bangor University, Wales, UK: The MIT Press, Jul. 2014, pp. 473–480. doi: 10.7551/978-0-262-32621-6-ch077.
[123] C. J. Headleand and W. J. Teahan, "Template Based Evolution," in *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*, in GECCO '13 Companion. New York, NY, USA: ACM, 2013, pp. 1383–1390. doi: 10.1145/2464576.2482718.
[124] John Speakman, "Evolving Natural Behavioural Phenomena with Template Based Evolution," Masters Thesis, Department of Computer Science, Bangor University, Bangor, Wales, 2015.
[125] R. V. Yampolskiy, "Why We Do Not Evolve Software? Analysis of Evolutionary Algorithms," *Evol. Bioinforma. Online*, vol. 14, Dec. 2018, doi: 10.1177/1176934318815906.
[126] M. O'Neill and L. Spector, "Automatic programming: The open issue?," *Genet. Program. Evolvable Mach.*, Sep. 2019, doi: 10.1007/s10710-019-09364-2.
[127] G. E. Moore, "Cramming More Components Onto Integrated Circuits," *Proc. IEEE*, vol. 86, no. 1, pp. 82–85, Jan. 1998, doi: 10.1109/JPROC.1998.658762.
[128] J. R. Koza, "Human-competitive results produced by genetic programming," *Genet. Program. Evolvable Mach.*, vol. 11, no. 3, pp. 251–284, Sep. 2010, doi: 10.1007/s10710-010-9112-3.
[129] S. Celis, *shanecelis/push-forth-dotnet*. (Sep. 19, 2019). C#. Accessed: Feb. 03, 2020. [Online]. Available: https://github.com/shanecelis/push-forth-dotnet
[130] Spector, Lee, "Push, PushGP, and Pushpop," Evolutionary Computing with Push. Accessed: Feb. 03, 2020. [Online]. Available: https://faculty.hampshire.edu/lspector/push.html
[131] C. Ryan, M. O'Neill, and J. J. Collins, Eds., *Handbook of Grammatical Evolution*. Springer International Publishing, 2018. doi: 10.1007/978-3-319-78717-6.
[132] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "DeepCoder: Learning to Write Programs," p. 20, 2017.
[133] "GitHub Copilot · Your AI pair programmer," GitHub. Accessed: Jul. 20, 2022. [Online]. Available: https://github.com/features/copilot
[134] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving Language Understanding by Generative Pre-Training".
[135] B. Cody-Kenny, E. Galván-López, and S. Barrett, "locoGP: Improving Performance by Genetic Programming Java Source Code," in *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, Madrid Spain: ACM, Jul. 2015, pp. 811–818. doi: 10.1145/2739482.2768419.
[136] V. Murali, L. Qi, S. Chaudhuri, and C. Jermaine, "Neural Sketch Learning for Conditional Program Generation," *ArXiv170305698 Cs*, Apr. 2018, Accessed: Jan. 22, 2020. [Online]. Available: http://arxiv.org/abs/1703.05698
[137] A. Solar-Lezama, "The Sketching Approach to Program Synthesis," in *Programming Languages and Systems*, vol. 5904, Z. Hu, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 4–13. doi: 10.1007/978-3-642-10672-9_3.
[138] M. Nye, L. Hewitt, J. Tenenbaum, and A. Solar-Lezama, "Learning to Infer Program Sketches," *ArXiv190206349 Cs*, Jun. 2019, Accessed: Jan. 22, 2020. [Online]. Available: http://arxiv.org/abs/1902.06349
[139] K. Martineau, "Toward artificial intelligence that learns to write code," MIT News. Accessed: Feb. 06, 2020. [Online]. Available: http://news.mit.edu/2019/toward-artificial-intelligence-that-learns-to-write-code-0614

[140] T. Blickle and L. Thiele, “A Comparison of Selection Schemes Used in Evolutionary Algorithms,” *Evol. Comput.*, vol. 4, no. 4, pp. 361–394, Dec. 1996, doi: 10.1162/evco.1996.4.4.361.
[141] D.-C. Dang, A. Eremeev, and P. K. Lehre, “Runtime Analysis of Fitness-Proportionate Selection on Linear Functions,” *ArXiv190808686 Cs*, Aug. 2019, Accessed: Jun. 29, 2021. [Online]. Available: http://arxiv.org/abs/1908.08686
[142] A. Bellos, *The Grapes of Math*. 2015. Accessed: Jul. 06, 2021. [Online]. Available: https://www.simonandschuster.com/books/The-Grapes-of-Math/Alex-Bellos/9781451640113
[143] B. Furht, Ed., “SIMD (Single Instruction Multiple Data Processing),” in *Encyclopedia of Multimedia*, Boston, MA: Springer US, 2008, pp. 817–819. doi: 10.1007/978-0-387-78414-4_220.
[144] richlander, .“NET Core guide.” Accessed: Feb. 12, 2020. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/core/
[145] R. Nordby, “bepuphysics,” bepuphysics. Accessed: Feb. 10, 2020. [Online]. Available: https://www.bepuentertainment.com
[146]. “NET Core - Cross-Platform Code Generation with Roslyn and .NET Core.” Accessed: Jan. 21, 2019. [Online]. Available: https://msdn.microsoft.com/en-us/magazine/mt808499.aspx
[147] “Compilation.cs.” Accessed: Apr. 04, 2019. [Online]. Available: http://source.roslyn.codeplex.com/#Microsoft.CodeAnalysis/Compilation/Compilation.cs,ec43f5a2c70b26f1
[148] B. Adams, “ASP.NET Core: Saturating 10GbE at 7+ million request/s,” Age of Ascent. Accessed: Feb. 10, 2020. [Online]. Available: https://www.ageofascent.com/2019/02/04/asp-net-core-saturating-10gbe-at-7-million-requests-per-second/
[149] “TechEmpower Web Framework Performance Comparison,” www.techempower.com. Accessed: Feb. 10, 2020. [Online]. Available: https://www.techempower.com/benchmarks/#section=test&runid=8ca46892-e46c-4088-9443-05722ad6f7fb&hw=ph&test=plaintext
[150] “Collectible assemblies in .NET Core 3.0 | StrathWeb. A free flowing web tech monologue.” Accessed: Feb. 21, 2019. [Online]. Available: https://www.strathweb.com/2019/01/collectible-assemblies-in-net-core-3-0/
[151] R. Petrusha, “MethodInfo Class (System.Reflection).” Accessed: Feb. 21, 2019. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.reflection.methodinfo
[152] R. Petrusha, “How to: Hook Up a Delegate Using Reflection.” Accessed: Feb. 27, 2019. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/how-to-hook-up-a-delegate-using-reflection
[153] “ASP.NET MVC Pattern | .NET,” Microsoft. Accessed: Jun. 07, 2021. [Online]. Available: https://dotnet.microsoft.com/apps/aspnet/mvc
[154] S. Prickett, “A Look at Server-Sent Events,” Medium. Accessed: Jun. 07, 2021. [Online]. Available: https://medium.com/conectric-networks/a-look-at-server-sent-events-54a77f8d6ff7
[155] bradygaster, “Introduction to ASP.NET Core SignalR,” Microsoft ASP.Net Documentation. Accessed: Jun. 07, 2021. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction

[156] “Best HTTP/2 | Network | Unity Asset Store.” Accessed: Jun. 07, 2021. [Online]. Available: https://assetstore.unity.com/packages/tools/network/best-http-2-155981?aid=1011l8NVc&utm_source=aff
[157] T. Helmuth, *thelmuth/program-synthesis-benchmark-datasets*. (Apr. 10, 2024). Shell. Accessed: Jul. 14, 2024. [Online]. Available: https://github.com/thelmuth/program-synthesis-benchmark-datasets
[158] R. Moll, “iJava - an online interactive textbook for elementary Java instruction: demonstration,” *J Comput Sci Coll*, vol. 26, no. 6, pp. 55–57, Jun. 2011.
[159] Y. Brun, E. Barr, M. Xiao, C. L. Goues, and P. Devanbu, “Evolution vs. Intelligent Design in Program Patching”.
[160] E. S. Henault, M. H. Rasmussen, and J. H. Jensen, “Chemical space exploration: how genetic algorithms find the needle in the haystack,” *PeerJ Phys. Chem.*, vol. 2, p. e11, Jul. 2020, doi: 10.7717/peerj-pchem.11.
[161] T.-P. Hong, C.-H. Chen, and F.-S. Lin, “Using group genetic algorithm to improve performance of attribute clustering,” *Appl. Soft Comput.*, vol. 29, pp. 371–378, Apr. 2015, doi: 10.1016/j.asoc.2015.01.001.
[162] C. Guo, Z. Yang, X. Wu, T. Tan, and K. Zhao, “Application of an Adaptive Multi-Population Parallel Genetic Algorithm with Constraints in Electromagnetic Tomography with Incomplete Projections,” *Appl. Sci.*, vol. 9, no. 13, Art. no. 13, Jan. 2019, doi: 10.3390/app9132611.
[163] A. Alajmi and J. Wright, “Selecting the Most Efficient Genetic Algorithm Sets in Solving Unconstrained Building Optimization Problem,” *Int. J. Sustain. Built Environ.*, vol. 3, Aug. 2014, doi: 10.1016/j.ijsbe.2014.07.003.
[164] Y. Oppacher, F. Oppacher, and D. Deugo, “Evolving java objects using a grammar-based approach,” in *GECCO*, 2009. doi: 10.1145/1569901.1570220.
[165] S. Wappler and J. Wegener, “Evolutionary Unit Testing Of Object-Oriented Software Using A Hybrid Evolutionary Algorithm,” in *2006 IEEE International Conference on Evolutionary Computation*, Jul. 2006, pp. 851–858. doi: 10.1109/CEC.2006.1688400.
[166] J. Speakman, “Evolving Source Code: Object Oriented Genetic Programming in .NET Core,” presented at the Society for the Study of Artificial Intelligence and Simulation of Behaviour Convention, Apr. 2019, pp. 16–19. Accessed: Feb. 10, 2020. [Online]. Available: http://aisb2019.falmouthgamesacademy.com/wp-content/uploads/2019/04/AISB-AI-AND-Games2019_proceedings.pdf
[167] O. Jadaan, C. R. Rao, and L. Rajamani, “Non-Dominated Ranked Genetic Algorithm for Solving Multi-Objective Optimisation Problems: NRGA,” *undefined*, 2008, Accessed: Jan. 07, 2022. [Online]. Available: https://www.semanticscholar.org/paper/NON-DOMINATED-RANKED-GENETIC-ALGORITHM-FOR-SOLVING-Jadaan-Rao/bb624311889b03dd78fca5d395ee34f106deebfa
[168] S. Forstenlechner, “Program Synthesis with Grammars and Semantics in Genetic Programming”.
[169] B. Berger, M. S. Waterman, and Y. W. Yu, “Levenshtein Distance, Sequence Comparison and Biological Database Search,” *IEEE Trans. Inf. Theory*, vol. 67, no. 6, pp. 3287–3294, Jun. 2021, doi: 10.1109/tit.2020.2996543.
[170] R. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st edition. Upper Saddle River, NJ: Prentice Hall, 2008.
[171] “PEP 8 – Style Guide for Python Code | peps.python.org,” Python Enhancement Proposals (PEPs). Accessed: Jul. 23, 2024. [Online]. Available: https://peps.python.org/pep-0008/

[172] N. Lorway, M. Jarvis, A. Wilson, E. Powley, and J. Speakman, “Autopia: An AI Collaborator for Gamified Live Coding Music Performances,” presented at the Society for the Study of Artificial Intelligence and Simulation of Behaviour Convention, Falmouth University, Apr. 2019, pp. 1–4. Accessed: Feb. 10, 2020. [Online]. Available: http://aisb2019.falmouthgamesacademy.com/wp-content/uploads/2019/04/AISB-AI-AND-Games2019_proceedings.pdf
[173] N. Lorway, J. Speakman, and E. Powley, “Autopia: An AI Collaborator for Live Coding Music Performances (Demo performance) on Vimeo.” Accessed: Feb. 10, 2020. [Online]. Available: https://vimeo.com/349044280
[174] S. Wilson, *Utopia*. (Jul. 26, 2021). SuperCollider. Accessed: Nov. 01, 2021. [SuperCollider]. Available: https://github.com/muellmusik/Utopia
[175] S. Luke, *Essentials of Metaheuristics*. Morrisville, N.C.: lulu.com, 2013.
[176] N. Lorway, E. Powley, and A. Wilson, “Autopia: An AI collaborator for live networked computer music performance,” presented at the 2nd Conference on AI Music Creativity (MuMe + CSMC), Institute of Electronic Music and Acoustics (IEM) of the University of Music and Performing Arts of Graz, Austria, Jul. 2021. Accessed: Jan. 11, 2022. [Online]. Available: https://aimc2021.iem.at/wp-content/uploads/2021/06/AIMC_2021_Lorway_Powley_Wilson.pdf
[177] M. Georgioudakis and V. Plevris, “A Comparative Study of Differential Evolution Variants in Constrained Structural Optimization,” *Front. Built Environ.*, vol. 6, 2020, Accessed: Feb. 17, 2022. [Online]. Available: https://www.frontiersin.org/article/10.3389/fbuil.2020.00102
[178] N. Hansen, “The CMA Evolution Strategy: A Comparing Review.” 2006.
[179] L. Spector, “An Essay Concerning Human Understanding of Genetic Programming,” in *Genetic Programming Theory and Practice*, R. Riolo and B. Worzel, Eds., Boston, MA: Springer US, 2003, pp. 11–23. doi: 10.1007/978-1-4419-8983-3_2.
[180] M. Chen *et al.*, “Evaluating Large Language Models Trained on Code,” Jul. 14, 2021, *arXiv*: arXiv:2107.03374. doi: 10.48550/arXiv.2107.03374.
[181] M. Bruch, M. Monperrus, and M. Mezini, “Learning from examples to improve code completion systems,” in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, in ESEC/FSE ’09. New York, NY, USA: Association for Computing Machinery, Aug. 2009, pp. 213–222. doi: 10.1145/1595696.1595728.