# Grammar-guided genetic programming

## Evolve programs in any context-free language for any quantifiable goal

**Abstract**

This document provides supplementary material for a Deep Funding proposal that aims to implement methods from grammar-guided genetic programming and then onboard the best-performing one as a service on SingularityNET's AI platform.

It contains theoretical background from optimization theory, explains how evolutionary algorithms practically solve a wide range of problems and why grammar-guided genetic programming is especially powerful when it comes to defining and traversing new search spaces. After this theoretical section, the status quo is discussed and how this project aims to improve upon it. Then follows an important section that lists the proposed developments and explains how they can be used to fulfill different roles on the SingularityNET platform and act in combination with other algorithms in OpenCog. Finally, it describes some example applications, where a preliminary prototype was used to successfully solve optimization problems from different domains in order to demonstrate the flexibility and capability of these methods.

In addition to this document, there is also a supplementary website with a gallery of additional examples. Both prototype code and the results from running it can be inspected in a single place with the help of Jupyter notebooks that were converted to HTML pages.

Website: https://robert-haas.github.io/g3p

## 1 Theoretical background

This Deep Funding proposal covers grammar-guided genetic programming algorithms, which are general-purpose optimization methods that can be applied to many optimization problems. Their special property is that they come with great flexibility regarding the definition of the search space, because they accept a user-provided context-free grammar to define the set of all candidate solutions and the neighborhood structure thereof. Practically it means that it is exceptionally easy to apply these methods to new problems, because in contrast to other evolutionary algorithms, dealing with a new search space does not require the adjustment of any internals, such as genotype structure, encoding scheme or search operators. This generality and flexibility enables these methods to play different roles in SingularityNET's network of AI services, which will be discussed in a later section. The following paragraphs will explain the technical terms used in this short characterization in a bit more detail to increase the clarity of central concepts:

**Optimization** means to search for the best element in a collection of available items, e.g. the shortest route in the set of all possible paths. Optimization is performed in two distinct steps:

1. Define an **optimization problem** by providing a) the **search space** containing all candidate solutions, e.g. all paths between two locations, b) an **objective function** that assigns a value to each item, and c) the **goal**, which is either minimization or maximization. The value provided by the objective function indicates how good each item is and thereby enables to look for the best one in the entire search space. E.g. if the item is a path, its length can be used as value and the search can look for a path with minimal length.
2. Solve the given problem by applying an **optimization method** that can operate on this kind of search space. There are, for example, many algorithms for numerical optimization that can operate in continuous Euclidean spaces and use gradient information of the objective function for efficient search. Many real-world problems, however, can not be tackled with these algorithms, for example because they have a discrete search space (sets, graphs, permutations) or the objective function is not analytic and its evaluation is too expensive to approximate gradients. From this point of view, optimization methods can be divided into **domain-specific** and **general-purpose** methods, depending on whether they can operate on only a few or on a wide range of different search spaces.

**Evolutionary computation (EC)** is a field that uses **evolutionary algorithms (EA)** to solve hard optimization problems. These algorithms have in common that they are all inspired by biological evolution, but with a greatly simplified view of the processes that actually occur in nature. The basic idea to start with an initial **population** of randomly generated **individuals**, whose **phenotype** is a candidate solution from the search space and whose **genotype** encodes that phenotype in some way, e.g. as a fixed-length bit string, which makes it easy to randomly modify individuals to generate new ones, e.g. by flipping some random bits. The central idea of evolutionary optimization is now that a population, which is effectively a set of candidate solutions, can be improved incrementally by a loop of **random variation** followed by **fitness evaluation** and **selection**, which creates one generation after another. Random variation is usually realized by applying mutation and crossover operators to genotypes, which creates new individuals, whose phenotype is determined by their genotype. Next their fitness value is determined by calling the objective function to rate their phenotype, which can be done in parallel for all individuals of a population, or even distributed on multiple machines. Finally they are filtered by a selection operator that favors individuals with better fitness values, which corresponds to higher quality of the candidate solutions. This iterative improvement of populations is continued until some stop criterion is met, e.g. when the solution quality is high enough, when a predefined number of generations is reached, or when a given amount of time has passed.

Today, evolutionary computation is a wide field with a large variety of approaches. Historically, it arose in different regions of the world independently, but mainly in four separate lines of research, which then gradually merged into the unified discipline of today. These branches were Genetic Algorithms (GA), Evolution Strategies (ES), Evolutionary Programming (EP) and Genetic Programming (GP). Of these branches, only genetic programming is concerned with the evolution of programs, and modern methods from that field are the topic of this proposal.

**Grammar-guided gentic programming (GGGP, G3P)** A more recent development in the field of evolutionary computation, and particular in genetic programming, is to use a formal grammar to define the search space. This means that each candidate solution is a string in the formal language defined by the grammar. It is desirable to do so, because grammars are ubiquitously used in computer science and software development. Especially **context-free grammars (CFG)** are the main device to specify the **syntax** of **general-purpose programming languages** (e.g. Algol, C, Python, Rust), **domain-specific languages** (e.g. HTML, JSON, protobuf) and user-designed **mini-languages** (e.g. describing geometric shape positions and colors). This means, a grammar defines which strings belong to a **formal language**, or in other words, what exactly a **syntactically valid program** in a language has to look like. Compilers and interpreters use these grammars, essentially to parse a given program into a syntax tree and then act on it, e.g. first transform the tree into a simpler one and then execute the commands it contains in its nodes.

The following is a list of grammar-guided genetic programming methods that shall be implemented in the course of this project. They were selected by different criteria, such as being foundational, popular or very recent developments, with the goal of covering a relevant and representative mix of methods from the field:

- Context-free grammar genetic programming (**CFG-GP**) [20]
- Grammatical evolution (**GE**) [9]
- Position-independent grammatical evolution (**piGE**) [10]
- Dynamically structured grammatical evolution (**DSGE**) [7]
- Weighted hierarchical grammatical evolution (**WHGE**) [1]

Claims about the relative performance of these methods, found in publications by different authors, are contradictory and rely on weak evidence, primarily because only a few and often too simple benchmark problems are used to measure their performance. For this reason, a more robust comparison will be done by collecting a larger set of suitable **benchmark problems**, applying all methods to them and evaluating the results with proper **statistical analysis**. This will be done according to guidelines set by the genetic programming community, which has recognized a long-standing benchmarking problem in its field [8, 21].

**SingularityNET and the Open Cognition Project (OpenCog)** SingularityNET is a response to increasing concentration of AI research, development and control in a few powerful corporations. The goal is to counteract this trend by transforming AI "from a corporate asset to a global commons" [2]. The short-term strategy is to empower AI developers by providing them with a novel, lucrative alternative for monetizing their software: in form of revenue-generating services on an open marketplace accessible to everyone, instead of selling it to a single best bidding entity. This is achieved by creating a **decentralized network of AI services**, built upon an open-source protocol and smart contracts. The long-term strategy is to onboard a diverse set of high-quality AI algorithms, which can not only be used individually, but also interoperate in a rapidly growing number of combinations to achieve increasingly complex goals. The vision for this incentivized network of interoperating algorithms is eventually enabling self-organization of AI services and ultimately "catalyzing the emergence and economic pervasiveness of self-modifying, self-improving, self-understanding artificial general intelligence" [2].

The development of the SingularityNET platform is complemented by the Open Cognition project, which develops the open-source software framework OpenCog for integrative AI, where a variety of machine learning and reasoning algorithms act together on a shared knowledge representation in form of a metagraph. This allows them to augment each others capabilities to overcome individual limits and achieve better performance, which is referred to as cognitive synergy. The goal of this project is to surpass the limits of narrow AI and create "general intelligence at the human level and beyond" [15]. OpenCog will also become active on SingularityNET, where its role can be thought of as analogous to that of the neocortex in a brain, i.e. providing higher-order reasoning capabilities to complement many other functionalities in a non-trivial way.

This proposal is based on the idea that an implementation of a performant grammar-guided genetic programming method can

1. **Play different valuable roles on the SingularityNET platform**, from standalone optimization service, over basic interactions with other services such as in hyperparameter optimization, to eventually becoming a flexible component in complex assemblies, possibly even plugging services together in creative ways to evolve pipelines or assist in the formation of cyclic pathways as observed in algorithmic chemistries.
2. **Act as an additional learning algorithm in OpenCog**, where different synergies with other algorithms are possible, e.g. by generating sets of creative candidate solutions in arbitrary spaces, where pattern mining [13] can identify useful sub-solutions, which in turn could inform the evolutionary search process to sample the space of candidates more intelligently through deliberately reusing those sub-solutions in the generation of new individuals.

## 2 Status quo and how it can be improved

**Existing implementations** In many cases, the authors of a GGGP method have provided a research article describing the algorithm and a reference implementation accompanying it. Unfortunately, there is currently little incentive in the academic community to invest a significant amount of time and resources into software quality, usability and maintenance. Consequently, the issue with the existing codebases is that they are written in different languages, structured in various ways and often with little focus on accessibility, modularity and maintainability. Sometimes it is even necessary to modify source code in order to apply a method to new examples. For these reasons, it is hard to utilize and extend these implementations directly, which also hinders proper comparison of their performance. One motivation of this project is to improve that situation by implementing relevant methods in a single place with shared abstractions, make them easily accessible by a simple, unified API and thereby also enable direct comparisons.

**Benchmarks** The genetic programming community has identified that it had a long-standing benchmarking problem [8]. In recent years, efforts have been made to improve the situation with measuring and reporting guidelines, for example by banning the use of improper toy problems when assessing the performance of a new method. The situation is still not resolved, however, and it is currently largely unknown which method will perform best on a specific problem in the context of a given use case [21]. This will be addressed in this proposal by collecting suitable benchmark problems and providing statistical analysis functions to evaluate the relative performance of the implemented methods.

**Prototype** I have implemented a proof-of-concept prototype that so far covers two methods (CFG-GP, GE) in an easily usable way, which is showcased on a website with Jupyter notebooks: https://robert-haas.github.io/g3p. A serious amount of work is still required to achieve good coverage of state-of-the-art methods and to develop the prototype into a coherent, robustly tested, well documented and easily maintainable open-source software package. Once this status is reached, the package can be used to create a reliable and performant service on the SingularityNET platform, thereby bringing general and flexible optimization capabilities to it.

**Other open-source projects by the author** I have released following Python packages, which may act as additional reassurance for the feasibility of this project:

- An interactive network visualization library: https://robert-haas.github.io/gravis-docs
- A package for visualizing OpenCog AtomSpaces: https://robert-haas.github.io/mevis-docs

The second package can be used to visualize Atomese programs evolved for OpenCog. An example can be found at the end of a Jupyter notebook: https://robert-haas.github.io/g3p/media/notebooks/atomese_symbolic_regression.html

Short background information: In OpenCog, both code and data are stored as Atoms in its knowledge representation, which is called AtomSpace. Mathematically the AtomSpace is a generalized hypergraph, also referred to as metagraph. This metagraph can be projected to a directed acyclic graph with two kinds of nodes, which is similar to projecting a hypergraph to a bipartite graph. This graph can then be used as input for interactive network visualizations, giving accurate insights into the contents of an AtomSpace.

**Open-source package and SingularityNET service** To my knowledge there is no service yet on the SingularityNET platform that provides G3P methods to other services, or that uses such an approach internally. Both situations will be addressed and changed by this proposal: on the one hand, a general-purpose optimization service will be provided that can be called by other agents; on the other hand, an open-source software package will be provided that can be incorporated in the internal processing of other projects.

# 3 Proposed developments and their utility

## 3.1 What will be created?

The goal is to implement grammar-guided genetic programming methods, find out which one performs best, and make it available as general-purpose optimization service on SingularityNET. The following is an overview of the main components that will be created to achieve this:

**Source code on GitHub** This includes the **main code** that implements five GGGP methods and supporting functionality, **test code** to check for its stability and correctness, **documentation** to explain its usage, and **project configuration files** to compress and distribute it as standard Python package that can be easily installed. All of that comes with a permissive open-source license, so anybody can use, extend and redistribute it freely. Ideas are most powerful when they are shared rather than owned!

**Distributed code on PyPI** This is a subset of the source code, wrapped into a package that can be uploaded to the **Python Package Index** (**PyPI**), which is the official repository for third-party libraries in the Python programming language. Once it is there, the code can be installed on other computers with **Python's default package installer** (**pip**) by a simple one line command.

**Benchmark collection** This will be implemented as subpackage within the main code. It contains **benchmark optimization problems** collected from literature and **statistical analysis functionality**, adhering to guidelines from the genetic programming community. It forms the basis for evaluating the relative performance of the implemented G3P methods by repeated runs on a diverse set of problems from different domains and proper statistical evaluation of the results.

**Documentation website** This is a website that explains how to install and use the package. It is achieved mainly by providing **introductory text**, **code examples** and an **API reference** that is semi-automatically generated from source code with a documentation tool called **Sphinx**. The generated website can then be hosted for free on **GitHub pages**. It will look similar to the websites I've created for my other open-source packages mentioned in the previous section.

**SingularityNET service** The central aim of this proposal is to onboard a novel AI service to the SingularityNET platform to support its growth. This will be achieved by combining 1) the best performing grammar-guided genetic programming method implemented in the proposed open-source package, and 2) the Daemon process and gRPC examples provided by the SingularityNET platform. SingularityNET's core components (CLI, SDKs, Daemon) are documented on its developer portal and allow anyone to "create, share, and monetize AI services at scale" [18].

The next section describes the preliminary interface of this service and how it can be used in increasingly sophisticated ways to solve hard optimization problems.

### 3.2 How can the package and service be used?

1. **Standalone usage**: The G3P service on SingularityNET can be used to solve optimization problems in a standalone fashion, i.e. without interacting with other services. The user provides all inputs via the service interface and gets the results:

   **Required inputs** that a user needs to provide to the service

   a) **Grammar** to define the search space, the set of all candidate solutions

      This is a text in BNF or EBNF format that defines a context-free grammar. The search space is the grammar's language, a set of valid strings.

   b) **Objective function** to define how good each candidate solution is

      This is a text containing a function definition. The function gets a candidate solution (string) and returns a fitness value (number).

      Since this is user-provided text that is interpreted as code, a sandboxing solution is needed to protect the host machine, either on the interpreter level (PyPy [19]) or operating system level (gVisor [5], NsJail [4]).

   c) **Objective** to decide whether to minimize or maximize the fitness value

   d) **Termination criterion** to define when the search is finished

   **Optional inputs** that a user may provide to the service for fine-tuning

   a) **Population size** to define the number of individuals in each generation

   b) **Search operators** to set how new individuals are generated and filtered

   c) **Other parameters** to influence further internals of the search algorithm

   **Outputs** that the service returns to the user

   **Best solution** that could be found in the search space so far

      This is a string from the grammar's language, for example an expression in Python, a program in Rust, or a string in a DSL for geometric shapes.

   **Fitness value** of the best solution

      This is a number that the objective function assigned to the best solution.

2. **Integrated usage**: The open-source package can be integrated in other packages or services. First it has to be installed, then a G3P method can be called from code and the search is performed with local computing resources, as seen on the example website. Alternatively, the search can be performed by calling the AI service on the SingularityNET platform to use remote computing resources. Which of these two options is better will depend on a) how costly the objective function is to evaluate in terms of time and memory, and 2) how large the population of individuals is that can be evaluated in parallel. Harder problems will have costly objective functions and rugged fitness landscapes that require larger populations. Therefore, assuming the service is hosted on powerful machines (e.g. compared to a laptop), using the service becomes increasingly attractive for a user who wants to solve a difficult real-world problem that requires significant and scalable computing resources.

3. **Cooperative usage**: There are two basic ways to interact with other services:

   a) **Another service calls the G3P service** as optimization engine

   The idea is that a special-purpose service can use this general-purpose service as flexible optimization engine. For example, a service that tries to find optimal trading strategies on user-provided data can do so by calling the G3P service. It needs to provide a grammar that defines a domain-specific language for trading strategies, together with an objective function that measures the quality of a candidate strategy, e.g. with proper backtesting.

   b) **The G3P service calls another service** for evaluating candidate solutions

   The objective function gets a candidate solution and needs to evaluate its quality. This could involve a call to another service, such as a demanding simulation that measures how well a constructed object behaves, or a machine learning model that achieves a certain error on a test set depending on which parameterization was chosen for it. The latter is an example of **hyperparameter optimization**, where the G3P service tries to find those input parameters for a model or service that maximize its performance.

   Technical remark: Since this mode requires the objective function to make calls to other services, it probably requires an adaption of the sandboxing approach. Another option is to provide a dedicated variant or mode of the G3P service, where a user can plug in a third-party service as quality measurement without having to write an (entire) objective function for it.

4. **Synergetic usage**: OpenCog is a framework for integrative AI where different machine learning and reasoning algorithms work together on a shared knowledge representation. A G3P method could act as such an algorithm, probably fulfilling a similar role as **MOSES** [12]. To my knowledge MOSES is currently able to evolve boolean and arithmetic expressions using advanced heuristics to guide the search. A G3P method, in contrast, can evolve programs in any context-free language, meaning that it is much more flexible but does not use as advanced operators.

   There is potential to **interact with other algorithms available in OpenCog** to achieve some forms of **cognitive synergy**. The following are some preliminary and unrefined ideas how that could be realized:

   - Improving the search space: A grammar induction algorithm [3] or machine reasoning algorithm [16, 14] could be used to construct or intelligently adapt a domain-specific language for a given task. It would define or improve the search space on which a G3P method can operate on, so it becomes easier to find good solutions in it.
   - Supporting the search operators: The pattern miner [13] could be used to find good sub-solutions in the set of candidate solutions that were evaluated so far, e.g. identifying useful sub-trees in the derivation trees of candidate solutions with high fitness. This information could be used to guide the evolutionary search, e.g. by injecting good sub-solutions into the variation operators.

- Replacing the objective function: In case of having a costly objective function, a machine learning algorithm could be used to approximate it from known input-output pairs, which works increasingly well with a rising number of evaluated candidate solution. The learned approximate objective function can then be used to quickly evaluate candidate solutions, e.g. to remove very bad ones without calling the actual costly objective function on them, or to enrich good solutions without actually evaluating all of them and thereby spending less time to move towards better regions in the search space.

5. **Emergent usage**: Once there are more AI services on SingularityNET and they start forming assemblies to achieve more complex goals, the roles of a general-purpose optimization service, which is able to operate on many search spaces, could become more intricate and exotic. Here are some examples that currently come to my mind:

   - The G3P service could be used to **search for optimal combinations of other AI services (pipelines, assemblies)** to solve a given task, which is related to automated machine learning or **AutoML**. This could be done by using a grammar that describes a space of AI service combinations and an objective function that evaluates how good a candidate combination is in solving a complex task. **AI-DSL** might support this idea in one way or another, e.g. by allowing to recognize and prune invalid combinations without having to test them, or by enabling to infer a grammar that describes a better search space consisting only, or largely, of valid combinations.
   - The G3P service could be used for **meta-optimization**, where one instance of the service is used to **search for an optimal grammar** that acts as input for another instance of the service. This would mean that the first instance searches through a space of grammars (e.g. all specializations of a general grammar), and the other uses that grammar in the usual fashion to search for an optimal expression in it for achieving some task. This could be a way towards **automatically building specialized AI services for solving specific tasks or for operating on specific data**.

     A closely related idea is to search for different neural network architectures, which are good at learning in certain domains, again with the potential to **spawn specialized models as new AI services on the platform**.
   - The G3P service could be used to act as fuel for an **algorithmic chemistry** by adding programs from the outside to a "flow reactor" of algorithms. For example, G3P could search for programs of a desired complexity or some other form criterion, add good solutions to the reactor and thereby either speed up its convergence to objects that produce one another, or move them towards higher levels of complexity. This could support the formation of different **autocatalytic cycles of program generators on SingularityNET**, which is a possible pathway towards emergence of an adaptive complex system of algorithms, with uncharted potential for self-modification and self-organization.

# 4 Preliminary results

This section covers five out of ten example problems that were successfully solved with a preliminary prototype. All of them can be found with source code and calculated results on the supplementary website https://robert-haas.github.io/g3p

## 4.1 Generative art: Evolve geometric shapes to resemble an image

**Task** Draw a number of semi-transparent geometrical shapes on a canvas and adapt their forms, positions and colors until the resulting image resembles a given target image. This task originated in 2008 in a blog article by Roger Johansson (formerly named Roger Alsing) [17], who used 50 semi-transparent nonagons (=polygons with nine edges) as shapes and a hill climber as search algorithm, though he called it genetic programming for unclear reasons. It was reproduced by other people in various forms since then, mostly using hill climbers or genetic algorithms, with different types of shapes (e.g. triangles, circles, polygons) and a varying number of them (e.g. 50, 100, 300).

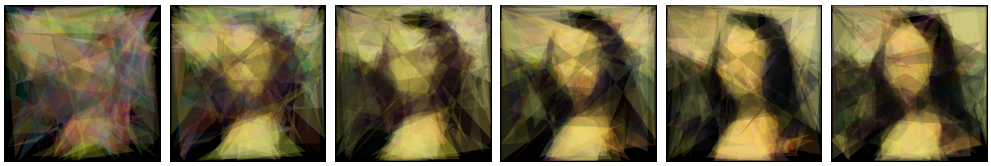
**Approach** To my knowledge there is no other publicly available demonstration of applying a G3P method to this task. A grammar can be used to define a simple domain-specific language, so that each string in it is a small Python program that when executed creates a canvas and draws shapes on it. The advantage of using a grammar to define the search space is its great flexibility: Changing a single rule allows to switch number or type of shapes, but also more complex choices become immediately available, e.g. allowing a mix of shapes, not fixing their number or splitting the task into multiple layers with different rules.

**Implementation** Jupyter notebook at https://robert-haas.github.io/g3p/media/notebooks/image_approximation.html

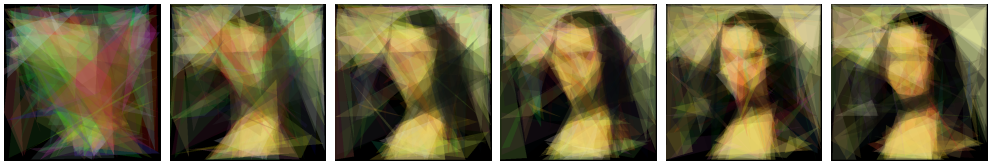
**Results** The following images show a reproduction of Roger Johansson's original example with 50 nonagons, but also very similar results with other types and numbers of shapes, which was achieved by minimally modifying one underlying grammar.

**Future directions** The idea could be extended from two to three dimensions, i.e. evolving 3D shapes in order to resemble up to three images from orthogonal viewing directions. This is realized in some art installations with physical objects together with the shadows they cast in x, y and z direction. Another example is the object on the cover of "Gödel, Escher, Bach" by Douglas R. Hofstadter [6] for which he coined the term ambigram [22].

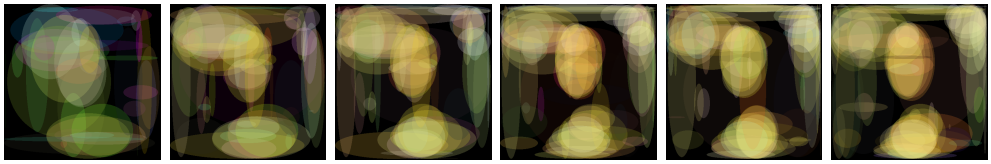
(a) Target image: Mona Lisa by Leonardo Da Vinci



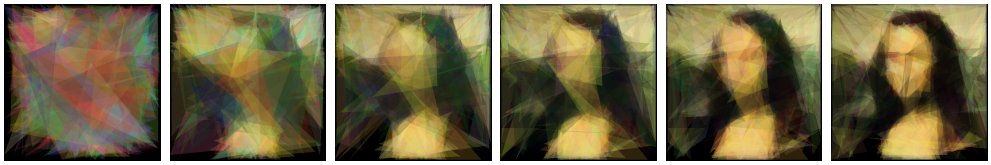
(b) Approximation with 50 nonagons



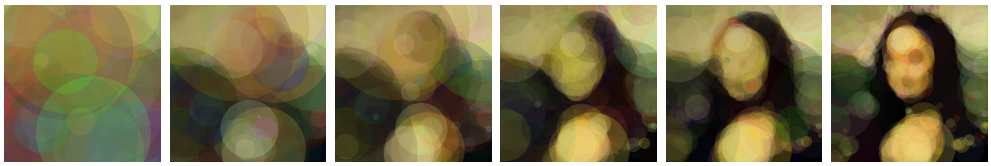
(c) Approximation with 150 triangles



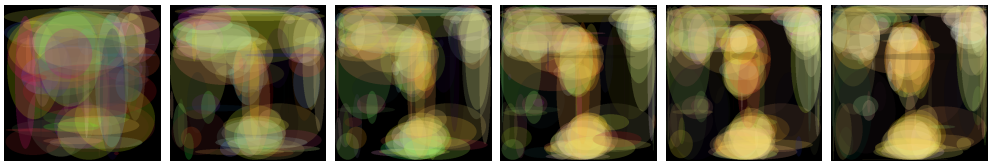
(d) Approximation with 150 ellipses



(e) Approximation with 300 triangles



(f) Approximation with 300 circles



(g) Approximation with 300 ellipses

Figure 1: Evolve small programs that draw semi-transparent geometrical shapes onto a canvas. The goal is that the resulting image shall closely resemble a given target image. Each row represents the use of a slightly different grammar and therefore optimization within another search space. This means that different numbers and types of shapes are drawn, and their positions and colors can be evolved to minimize the pixel-wise difference from a target image. From left to right the number of generations and elapsed computation time increases and therefore the solution quality gradually improves.

### 4.2 Control problems: Evolve agents to solve OpenAI Gym tasks

**Task** OpenAI Gym [11] is a collection of problems for testing and comparing reinforcement learning algorithms. In this machine learning paradigm, an agent is operating in an environment, which it can partially or fully observe, and in which it can take actions with the goal to maximize a cumulative reward over time. The examples tackled here are classicial problems from control theory, referred to as CartPole-v1, MountainCar-v0, Acrobot-v1 and Pendulum-v0 in OpenAI Gym. The agents to be controlled are a cart, a vehicle and two different pendulums. The environments are relatively simple and can be solved by policies that do not keep memories of past states, i.e. acting just on current observations. There is, however, no reason why G3P could not be applied to harder environments, which require policies that incorporate episodic memory.

**Approach** To my knowledge there is no other publicly available demonstration of applying a G3P method to these tasks. A grammar can be used to define a simple language, so that each string in it is a small Python program that when executed creates an Agent class that can be used to receive perceptions from the environments and take according actions to achieve the predefined goal. The advantage of using a grammar to define the search space is again its great flexibility: Changing the structure of the Agent allows to search for different kinds of solutions and check which approach works well on different environments. Here a grammar was used, which essentially evolves an algebraic expression that uses the observable quantities as input variables and the action to take as output variable. This means the concrete values coming from observing the current state of the environment are plugged into the expression to receive a number that encodes the action to take in response. For example, in the pole balancing task, whenever the pole starts leaning towards the left, this is reflected by the value of different observables and via a suitable algebraic expression it results in the action to move the cart to the left and thereby bring the pole back to an upright position.

**Implementation** Four Jupyter notebooks containing "OpenAI Gym" and the environment name in their title at https://robert-haas.github.io/g3p/

**Results** The following images depict solutions for four OpenAI Gym environments that come from classical control theory. First the text for program is shown that was evolved to implement a candidate agent with good fitness. Then a few screenshots are shown from simulations that let this agent act in its environment, where it is able to solve the task successfully on different random starting conditions.

**Future directions** G3P methods can be applied to harder problems in OpenAI Gym and other reinforcement learning platforms. In this way, the limit of using algebraic expressions for controlling an agent can be explored, and once reached, a transition to evolved agents that use memories can be attempted. Another direction is to combine different algorithms, such as evolutionary learning and temporal reasoning, to solve tasks that require both creativity and planning.

```
class Agent:
    def decide(self, observation):
        position, velocity, angle, angle_velocity = observation
        output = ((1.00+(1.92*angle))+(1.12*angle_velocity))
        action = int(output)
        return action
```

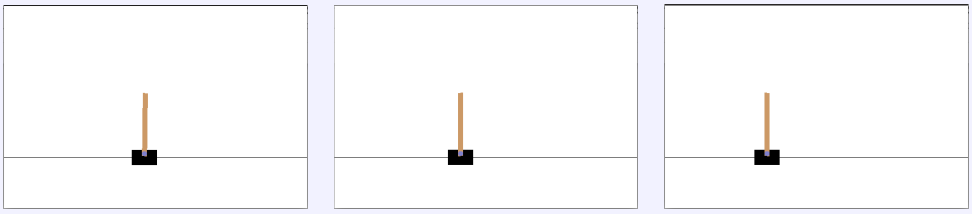


Figure 2: CartPole-v1 solved by an evolved Python program that implements an agent. The aim is to move a cart (black) so that it balances a pendulum (brown) without moving too far from the center. The agent observes the current position and velocity of the cart, as well as angle and velocity of the pole. It can act by pushing the cart to the left (value 0) or to the right (value 1).

```
class Agent:
    def decide(self, observation):
        position, velocity = observation
        output = (4.59/(3.35*velocity))
        action = 0 if output < 1.0 else 2
        return action
```

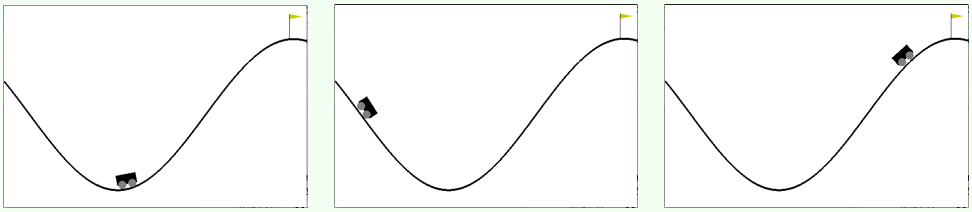


Figure 3: MountainCar-v0 solved by an evolved Python program that implements an agent. The aim is to drive a car up the right hill, but its engine is not strong enough, so it needs to build up momentum first. The agent observes the current position and velocity of the car. It can act by pushing the car to the left (value 0), applying no push (value 1), or pushing it to the right (value 2).

```
class Agent:
    def decide(self, observation):
        x0, y0, x1, y1, v0, v1 = observation
        output = (6.72*((v1*((5.01-0.80)**(5.94*(0.91*(v1+1.32)))))*((y0/v1)+5.87)))
        action = 0 if output < 1.0 else 2
        return action
```

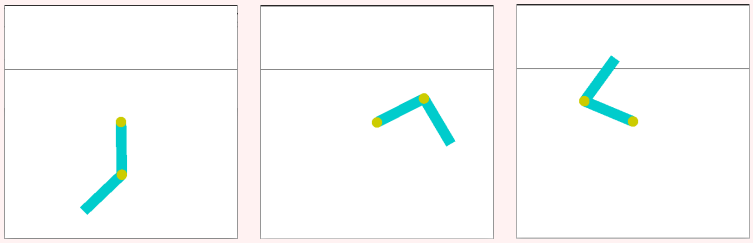


Figure 4: Acrobot-v1 solved by an evolved Python program that implements an agent. The aim is to swing the lower part of a two-link robot up to a given height, much like a gymnast. The agent observes current positions and velocities of the joints. It can act by applying positive torque (value 0), no torque (value 1), or negative torque (value 2) only to the joint between the two links.

```
class Agent:
    def decide(self, observation):
        x, y, angle_velocity = observation
        output = ((((2.05*x)-x)*((x-6.40)-(angle_velocity/y)))/y)
        action = [min(max(output, -2.0), 2.0)]
        return action
```

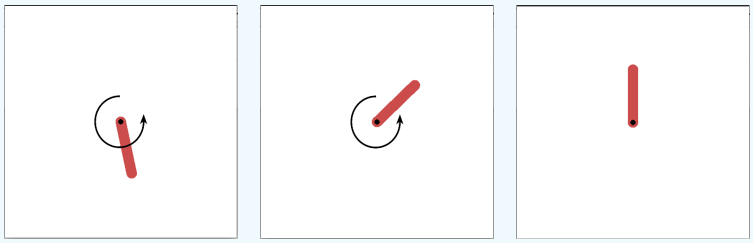


Figure 5: Pendulum-v0 solved by an evolved Python program that implements an agent. The aim is to swing up a frictionless pendulum and keep it standing upright there, starting from random position and velocity. The agent observes the current position and velocity of the pendulum. It can act by applying limited torque to the joint (continuous value between -2 to +2)

.

## References


[1] Alberto Bartoli, Mauro Castelli, and Eric Medvet. "Weighted Hierarchical Grammatical Evolution". In: *IEEE Transactions on Cybernetics* 50.2 (Feb. 2020), pp. 476–488.

[2] SingularityNET Foundation. *SingularityNET: A Decentralized, Open Market and Network for AIs*. 2019. URL: https://public.singularitynet.io/whitepaper.pdf.

[3] Alex Glushchenko et al. "Programmatic Link Grammar Induction for Unsupervised Language Learning". In: *Artificial General Intelligence*. Ed. by Patrick Hammer et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, July 2019, pp. 111–120.

[4] Google. *gVisor*. 2022. URL: https://github.com/google/gvisor.

[5] Google. *NsJail*. 2022. URL: https://github.com/google/nsjail.

[6] Douglas R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books Inc., 1979.

[7] Nuno Lourenço et al. "Structured Grammatical Evolution: A Dynamic Approach". In: *Handbook of Grammatical Evolution*. Ed. by Conor Ryan, Michael O'Neill, and JJ Collins. Cham: Springer International Publishing, 2018, pp. 137–161.

[8] James McDermott et al. "Genetic Programming Needs Better Benchmarks". In: *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*. Ed. by Terence Soule and Jason H. Moore. GECCO '12. New York, NY, USA: ACM, July 2012, pp. 791–798.

[9] Michael O'Neill and Conor Ryan. "Grammatical Evolution". In: *IEEE Transactions on Evolutionary Computation* 5.4 (Aug. 2001), pp. 349–358.

[10] Michael O'Neill et al. "pi Grammatical Evolution". In: *Genetic and Evolutionary Computation – GECCO-2004, Part II*. Ed. by Kalyanmoy Deb et al. Vol. 3103. Lecture Notes in Computer Science. Seattle, WA, USA: Springer-Verlag, 26-30 June 2004, pp. 617–629.

[11] OpenAI. *Gym*. 2016. URL: https://gym.openai.com.

[12] OpenCog Foundation. *Meta-optimizing semantic evolutionary search (MOSES)*. 2021. URL: https://wiki.opencog.org/w/Meta-Optimizing_Semantic_Evolutionary_Search.

[13] OpenCog Foundation. *Pattern miner*. 2020. URL: https://wiki.opencog.org/w/Pattern_miner.

[14] OpenCog Foundation. *Probabilistic logic networks (PLN)*. 2018. URL: https://wiki.opencog.org/w/Probabilistic_logic_networks.

[15] OpenCog Foundation. *The Open Cognition Project*. 2021. URL: https://wiki.opencog.org/w/The_Open_Cognition_Project.

[16] OpenCog Foundation. *Unified rule engine (URE)*. 2019. URL: https://wiki.opencog.org/w/Unified_rule_engine.

[17] Roger Johansson. *Genetic Programming: Evolution of Mona Lisa*. 2008. URL: https://rogerjohansson.blog/2008/12/07/genetic-programming-evolution-of-mona-lisa.

[18] SingularityNET Foundation. *AI Dev Community*. 2022. URL: https://dev.singularitynet.io.

[19] The PyPy Project. *PyPy's sandboxing features*. 2022. URL: https://doc.pypy.org/en/latest/sandbox.html.

[20] Peter A. Whigham. "Grammatically-based genetic programming". In: *Proceedings of the Workshop on GP: From Theory to Real-World Applications*. Ed. by Justinian P. Rosca. Tahoe City, California, USA, Sept. 1995, pp. 33–41.

[21] David R. White et al. "Better GP benchmarks: community survey results and proposals". In: *Genetic Programming and Evolvable Machines* 14.1 (Mar. 2013), pp. 3–29.

[22] Wikipedia. *Ambigram: 3-Dimensional ambigrams*. 2022. URL: https://en.wikipedia.org/wiki/Ambigram#3-Dimensional_ambigrams.