POLISH-JAPANESE ACADEMY OF INFORMATION TECHNOLOGY

Department of Computer Science

Department of Mechanics, Information Technology and Robotics
Robotics

Volodymyr Yanushevskyi
S16940

Evolutionary Program Synthesis

Bachelor of Science

Supervisor: MSc. Eng. Piotr Gnyś

Warsaw, February, 2021

POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Informatyka

Katedra Mechaniki, Informatyki i Robotyki

Robotyka

Volodymyr Yanushevskyi
S16940

Ewolucyjna Synteza Programów

Inżynierska

Promotor: Mgr. Inż. Piotr Gnyś

Warszawa, Luty, 2021

Abstract

This thesis discusses an evolutionary program synthesis approach of navigating the search space of possible programs. Search process is equipped with a read-eval-print-loop library (REPL) which executes written programs to expose its semantics. Paper applies the concept for inferring 2D graphics. Furthermore, some limitations of the approach are discussed, which are inherent to the nature of evolutionary methods.

Keywords: Program synthesis, Picture generation, Evolutionary methods

Abstract

Niniejsza praca dyplomowa omawia podejście do ewolucyjnej syntezy programów, polegające na nawigowaniu w przestrzeni poszukiwań możliwych programów. Proces wyszukiwania jest wyposażony w bibliotekę REPL (read-eval-print-loop), która wykonuje napisane programy w celu ujawnienia jego semantyki. W artykule zastosowano koncepcję wnioskowania graficznego 2D. Ponadto omówiono pewne ograniczenia tego podejścia, które są nieodłącznie związane z naturą metod ewolucyjnych.

Słowa kluczowe: Synteza programów, Generowanie obrazów, Metody ewolucyjne

Acknowledgements

I would like to thank my supervisor, Piotr Gnyś, for assistance and guidance with this topic, which helped avoiding a considerable amount of pitfalls along the way.

Contents

1 Introduction

In the recent years the world has observed a rise in automation, which freed a lot of labor from repetitive menial jobs with some professions, such as Dictaphone Operator , even going extinct [27]. One of the aspects why it was achieved is because the final product (or action) was known, and a set of operations that would lead to such a result were either fully or almost fully determined so it can be automated. If we recall the definition of Computer Programming, which is stated as: "Computer programming is the process of designing and building an executable computer program to accomplish a specific computing result"[8], it is apparent that automation should be happening. Although currently there are 26.4M programmers around the world, with 45M expected in 2030, which suggests an idea that developers are to stay [26]. It is so because, though a final specification is known, the search space of a sequence of programs, that would yield the desired result, is vast. This paper presents an evolutionary approach of writing programs with a concrete example of drawing 2D graphics as a sequence of instructions to match a specification (image) provided.

2 Evolutionary Algorithms

2.1 Overview

Evolutionary Algorithms is a popular generic population-based metaheuristic optimization subset of evolutionary computation [30]. When analytic solutions are impossible to find, probabilistic models give high-quality solutions for optimization problems. An EA uses mechanisms inspired by real-life biological evolution, such as selection, crossover, mutation [9]. In the beginning of the algorithm an initial population is selected and in an iterative way the fitness of every individual is calculated, the fittest are selected for further reproduction with occasional mutations.

2.2 History

Descriptions of evolutionary processes for computer problem solving first appeared in articles of Friedberg [16]. At the same time, Bremermann presented first attempts to apply simulated evolution onto numerical optimization problems [4]. Also, Bremermann developed some early evolutionary algorithm theory about optimality of mutation probability [5].

In the early 1960s, in order to create systems that are robust, and have an ability to respond quickly to the changes of environment and unanticipated events, simple models of biological evolution were chosen as a starting point [9]. The main reason for that is it biological evolution was seen to capture the ideas above nicely via notions of survival of the fittest and continues change of generations.

Bagley's thesis presented some early experimental work with the selection methods [1]. During this time Holland continued his research, which resulted into pivotal book 'Adaptation in Natural and Artificial Systems' [19]. Due to lack of computational power, it was hard to check theoretical part with experimental. De Jong's thesis broaded the line of study by combining both theoretical and experimental analysis of effects of population size, crossover, and mutation [20]. This study gave a strong sign that this approach had a significant potential for solving complex optimization problems.

Subsequent workshops, discussions and conferences led to a rising popularity and more research effort put in. Appearance of several books helped establishing the topic among broad audience of scientists and engineers [10].

The period from 1990 to the present has been characterized by tremendous growth and diversity of the community, publications of new books on the topic, and growing list of research papers. New GA algorithms continue being developed for a wide range of problems [11].

2.3 Representation of the data

One of the most important criteria for getting desired results is a good genetic representation. Representations can encode the appearances, behaviours, traits. A candidate solution in a computer algorithm is called an individual. An individual is characterized by a set of parameters called genes. Genes joined as a together form a chromosome, which encodes an individual. A set of individuals is called a population. Usually, binary values of 0's and 1's are used for encoding.

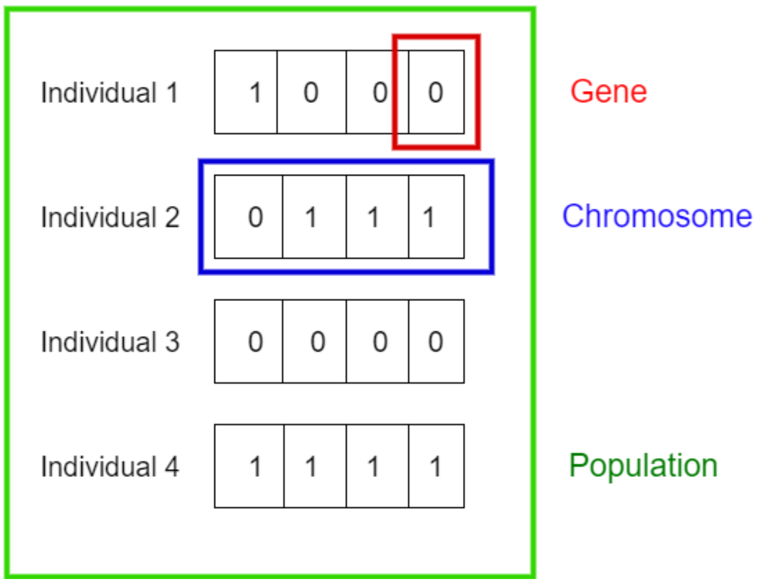


Figure 1: Genes, individuals and population

2.4 Main loop of the algorithm

The algorithm starts with a randomly generated initial population and then it enters an iterative loop, as described in the Figure 2.

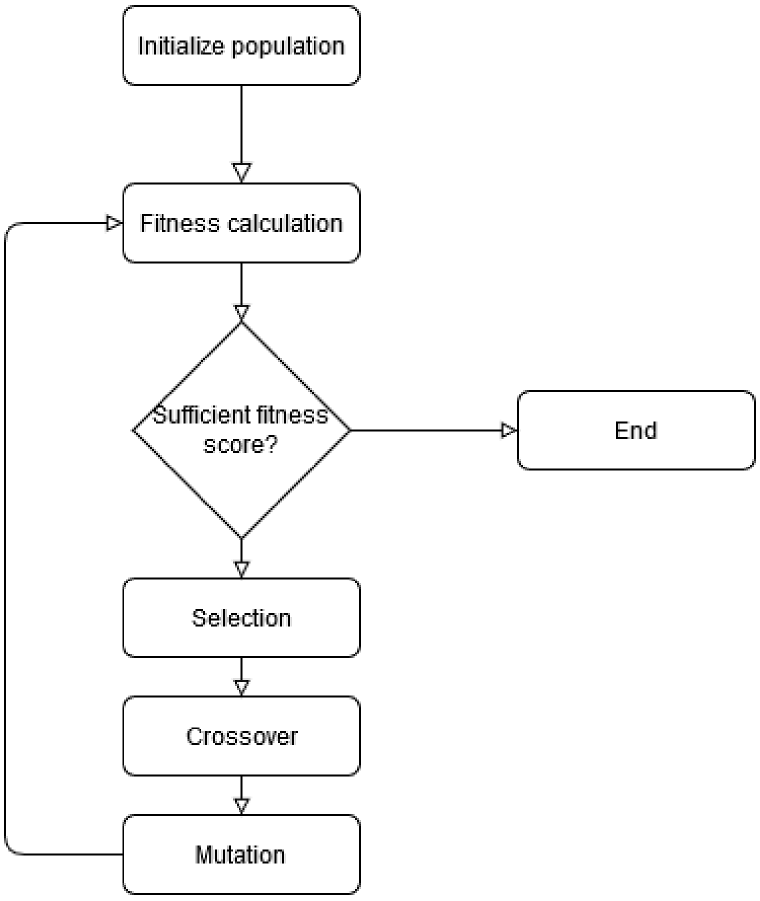


Figure 2: Main loop of the algorithm

2.5 Initialization of initial population

For initial population to be generated, some parameters are to be specified. Usually those include, but not limited to:

- length of a chromosome
- population size

Length of a chromosome defines how many bits of representation the chromosome should have. The size of the respective population is the number of individuals in the algorithm.

2.6 Fitness function

Fitness function is playing an important role in EA. This function represents a specific requirement and can determine how close the solution is to the optimal one. Therefore, it defines what an improvement is and, also, grants the ability to compare two different solutions to understand which one is better fit for the final goal.

From technical point of view, it represents a solution to a task, which is to be achieved. It assign a qualitative score to the particular phenotype of a selected representation.

The fitness function is problem-dependent, however, there are some generic requirement for selecting it that should be satisfied by any function of that kind [23]:

- The fitness function should be clearly defined. The reader should be able to clearly understand how the fitness score is calculated.
- The fitness function should be implemented efficiently. If the fitness function becomes the bottleneck of the algorithm, then the overall efficiency of the genetic algorithm will be reduced.
- The fitness function should quantitatively measure how fit a given solution is in solving the problem.
- The fitness function should generate intuitive results. The best/worst candidates should have best/worst score values.

Example of Fitness function:

To illustrate the process of defining the fitness function, let the example problem be finding solutions for x so that it would equal t in the following equation $3x^2 - 3x + 1 = t$, $0 < x < 31$. First of all, our potential fitness function should assign a qualitative score to a particular solution. Therefore, we can introduce a fitness function which represents a deviation from the original equation. It would have look like: $|3x^2 - 3x + 1 - t|$, which bears a meaning of 'the smaller the result, the closer our potential solution to a right one.'

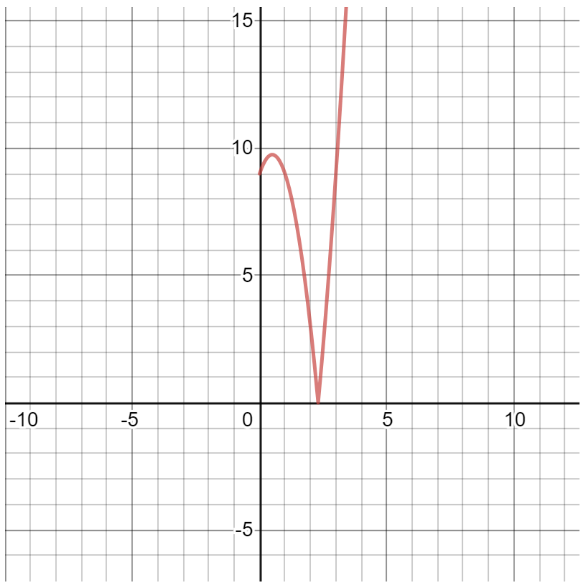


Figure 3: Graph of $|3x^2 - 3x + 1 - t|$, $0 < x < 31$

Usually, we want to either maximize or minimize the fitness values, and mathematically minimizing $|3x^2 - 3x + 1 - t|$ is equal to maximizing the $\frac{1}{|3x^2-3x+1-t|}$

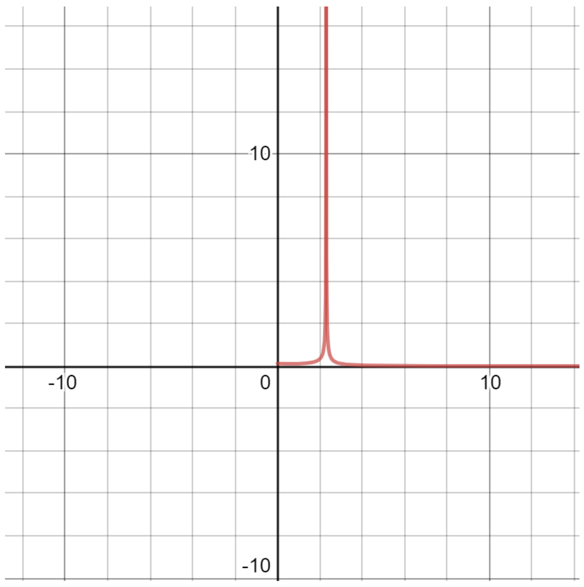


Figure 4: Graph of $\frac{1}{|3x^2-3x+1-t|}$, $0 < x < 31$

2.7 Selection

After deciding upon the encoding, the next important step is to define a mechanism of choosing individuals out of the population that will be used to create offsprings. This process of selection gives more value to fitter individuals in the population to increase the probability that children will be even fitter.

Balancing selection is one of the highest priorities, as having a strong selection will yield suboptimal children that will take over the whole population reducing diversity, while too weak selection will result in slow evolution. Numerous selection mechanism with its pros and cons are described below.

2.7.1 Fitness Proportionate Selection

One of the first used selections were Fitness Proportionate Selection [19]. Main idea of this type of selection is to define probability as individual's fitness over average population fitness.

The most common and easy method for this type of selection is a Roulette Wheel. In this form of mechanism, each individual is assigned a 'slice' of circular roulette wheel which size is proportionate to the fitness of that particular individual. The wheel can be spun N times to select N parents.

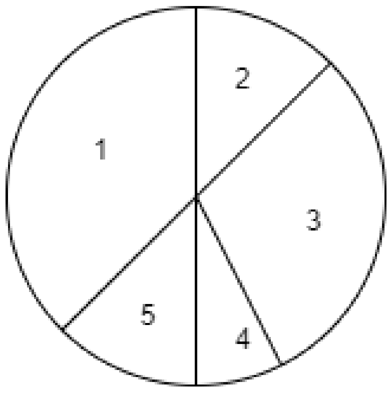


Figure 5: Individuals represented in a roulette

To illustrate the algorithm, some defined order of the population is defined from 1 to L, so that cumulative probability distribution can be calculated in a form of a list $a = [a_1, a_2, .., a_L]$ with $a_i = \sum_{k=1}^{i} P(k)$. The pseudocode for this selection is provided below.

```
/* Given k individuals are needed to be selected */
selected = 0
WHILE selected <= k:
    r = random value from uniform distribution [0,1]
    i = 0
    WHILE a[i] < r:
        i += 1
    mating_pool.add(parents[i])
    selected += 1

return mating_pool
```

Despite being simple, this approach has a relatively slow convergence rate, compared to other methods and an actual number of offsprings allocated to an individual is often far from its expected value [32]. So whenever a sample of k elements is to be drawn from a population, another fitness proportionate method is preferred: Stochastic Universal Sampling (SUS), which combats the issue of minimizing the spread of actual values, given the expected value [2].

In SUS, the individuals are selected in a fitness-proportionate manner, but in a way so that fit individuals are picked at least once.

```
/* Given k individuals are needed to be selected */
selected = 0
r = random value from uniform distribution [0,1/k]
WHILE selected <= k:
    WHILE r <= a[i]:
        mating_pool.add(parents[i])
        r += 1/k
        selected += 1
    i += 1

return mating_pool
```

There is one disadvantage of the methods presented above - is that they fail when fitness scores are nearly identical. If our fitness ranges from 0 to 10 and most of our individual mark at 9.97, 9.98 and 9.99, we would want to select 9.99 individual, but in fitness proportionate methods, all of the probabilities will be nearly uniform.

2.7.2 Tournament Selection

In order to combat the issue of sensitivity to the almost equal values of fitness scores, a non-parametric method called 'Tournament Selection' is presented. The main idea of this method is to throw away the fitness values and consider the ranking. Tournament selection also is useful because it doesn't require any global knowledge of the population, therefore it's faster and simpler.

```
/* Given k individuals are needed to be selected*/

```

```
t = tournament size
best = random individual from population with replacement
FOR i from 2 to t:
    contestant = random individual from population with replacement
    IF fitness(contestant) > fitness(best):
        best = contestant
return best
```

The best individual out of t randomly selected ones from the population is selected. The probability that a particular individual will be selected is dependent on such factors [13]:

- Its comparative rank in the population. This can be calculated without the knowledge of the full population
- The tournament size k. The larger the value of k, the bigger the average fitness of the selected winner, hence the k increases the selection pressure
- The probability that the most fit individual will be selected as a winner. If $p = 1$, then the tournament is deterministic, though stochastic versions are also possible, if $p < 1$ is set. The lower p is, the lesser the selection pressure is.
- Whether individuals are chosen with or without replacement. If method without replacement is chosen, it's no longer deterministic and it's becoming possible for least-fit individual to be selected.

It was shown that binary tournaments ($t = 2$) are preferred for best results [17]. Though, for some problems it's more common to have more selections, with values for $t = 7$ [22].

2.7.3 Elitism

Elitism is an addition to selection algorithms that forces to retain some number of individuals at each generation [25]. These individuals are called elites. This approach has an exploitative property that can cause premature convergence.

To combat this issue, some higher numbers in a mutation noise or more loose selection pressure might be needed. Elitism is a popular technique that is essential in multiobjective algorithms, which are mostly elites. Elitist selections do better than the ones that do not. Moreover, the quality of solutions which employ elitist selections monotonically increases over time. Without this form of selection, it's possible to lose the best chromosomes due to stochastic errors [3].

2.7.4 Steady-State Selection

Most of EA algorithms are 'generational' in a sense that each loop of the algorithm a new generation is created as offsprings of the parent generation. Some approaches, as Elitist strategy discussed above, preserve a number of parent individuals into the next generation. In Steady-State Selection only a fraction of population gets replaced, a small minority that is the least fit. The new individuals are a result of a crossover and mutation between the fittest in that generation.

This type of strategy has 2 important features. Firstly, it uses half of the memory that a typical generational algorithm would use, because there's no need to store 2 generations at a time, since there's only one. Secondly, the algorithm gets more exploitative, which may result into an algorithm converging to the copies of the most successful individuals. One way to counter this is to implement a $selectForDeath$ method, that would select random individuals for a death. Usual implementation of that approach replaces only 2 individuals each generation, but it can be extended to replace any percentage all at once. These types of methods are known as Generation Gap Algorithms [29]. As the replace percentage approaches 100, we get a normal generational algorithm.

2.7.5 Summary

Fitness Proportionate Selection is a simple approach, which has a rather slow convergence and is badly performing when fitness values are almost identical.

Tournament Selection offers a faster approach which also deals with the drawbacks of the Fitness Proportionate Selection, but is more complicated.

Elitism is an addition to the selection algorithms, that improves the general quality of soltuions.

Steady-State Selection is an idea of a majority of population should survive to the next generation

2.8 Crossover

A binary variation operator is called crossover(recombination). It is an operator to stochastically generate new offsprings with the genetic information of its respective parents: the choices of which parts of which parent are combined and how it is done is random.

The principle and idea behind this is simple. Throughout the human history, it's been successfully done to plant and livestock to create species that are superior in any way that was needed or to delete traits that are unwanted. Since most of the life on the Earth reproduces sexually, the concept of crossover is a dominant form of reproduction.

There are several popular approaches to define this variation operator and the ones presented below are discussed mostly in a context of a bit-string encodings.

- Single-point crossover

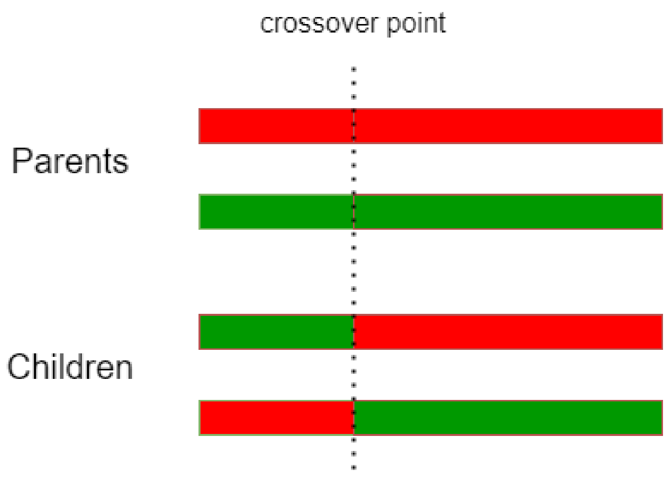


Figure 6: Single-point crossover

A point on on both parents' chromosomes are selected as 'crossover point'. Bits to the left of the crossover point are swapped between two parents.

- Two-point and k-point crossover

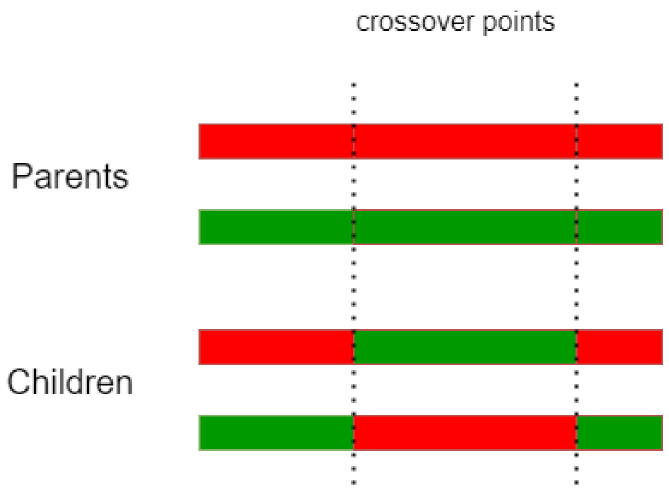


Figure 7: Two-point crossover

In two-point crossover, 2 crossover points are selected and the bits between these 2 points are swapped between the parents.

In the k-point crossover case, is the extension of two-point method to k points. Bits between two subsequent points are swapped between parents.

- Uniform crossover

  Previous two crossover operation worked under a concept of dividing a gene into parts and combining them to produce offsprings. Uniform crossover treats every gene as an independent entity and decides upon gene of which parent a child would get. A probability of whether to choose from parent A or from parent B is usually $p = 0.5$.

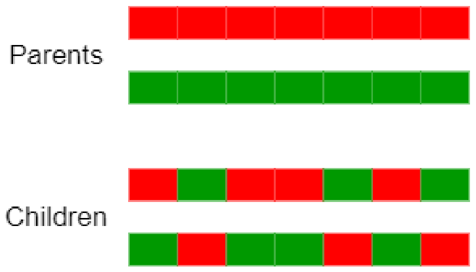


Figure 8: Uniform crossover

Selecting which approach to use may be particularly difficult since there are a lot of factors in play: positional bias, disruption potential, etc. Most general observation is that two-point crossover and uniformed crossover with p of about 0.7-0.8 work better [25].

Needless to say, that the question of why recombination is useful in real-life biology (if it indeed is) - is an open question still [25].

2.9 Mutation

The main purpose of mutation is to introduce diversity into the heuristics. This helps avoiding the local minima by preventing the population becoming too similar. By setting the probability of the mutation low, the better results can be reached. A good algorithm would have a good balance between exploration and exploitation. Exploration means a search of the searching space, while exploitation means a concentration on one point in the space. Most popular method of this genetic operator is flipping random bits in the chromosome.

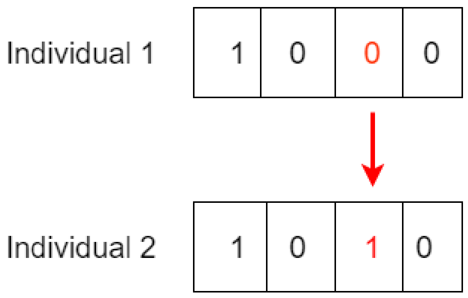


Figure 9: Mutation

3 2D Graphics

3.1 Overview

Computer graphics has become ubiquitous all around the world with the advent of computers. It has gone a long way since it's creation in the 1950s as a visualisation tool for scientists and engineers in government and corporate research centers such as Bell Labs and Boeing [24]. Computer graphics research is still continued all around the world, as research and development departments of entertainment and production companies became more and more interested in creating synthetic digital reality.

Not only it is used for entertaining purposes, the importance of computer graphics lies in its application in other field as well [31].

- In engineering, the possibility of visualising the shapes has proven to be indispensable. It has reduced the number of human-hours for production of clay models for prototyping purposes.
- In medical field, the advances of computer tomography and magnetic resonance imaging allowed physicians to take X-rays of the human body.

3.2 Computer-Aided Design

Computer-Aided Design (CAD) can be defined as a use of computer programs in order to aid in the creation, modification, analysis or optimization of design [28]. CAD software consists of computer programs which implement computer graphics.

Modern CAD systems are used for interactive computer graphics (ICG). ICG is a system in which computer is employed to create and transform data in the form of pictures. A designer creates an image by entering the commands to call software's functions, which (in most systems) construct an image out of geometric shapes, such as circles, lines, etc. Through this set of requirements, a final image is created.

In this form of cooperation, the human creates parts that is more suitable to human intellect and skill, while the machine takes over what it does more efficient.

3.3 Constructive Solid Geometry

In Costructive Solid Geometry (CSG), simple objects, known as primitives, are combined by using Boolean operators Union, Intersection, Difference and additionally by using Rigid transformation such as Scale, Rotation, Translation, Shear. An object is stored in a form of a tree with operators as nodes and primitives as leaves [15]. Since Boolean operations are not commutative, the edges of the tree are ordered.

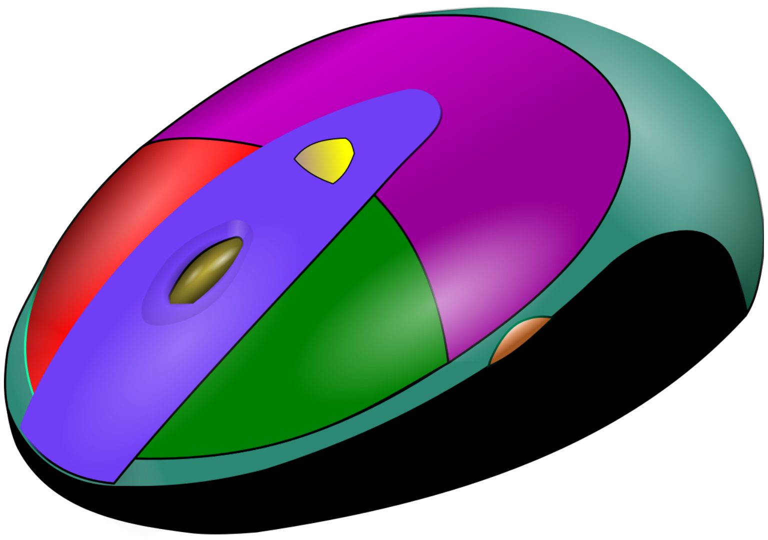


Figure 10: CAD model of a computer mouse [12]

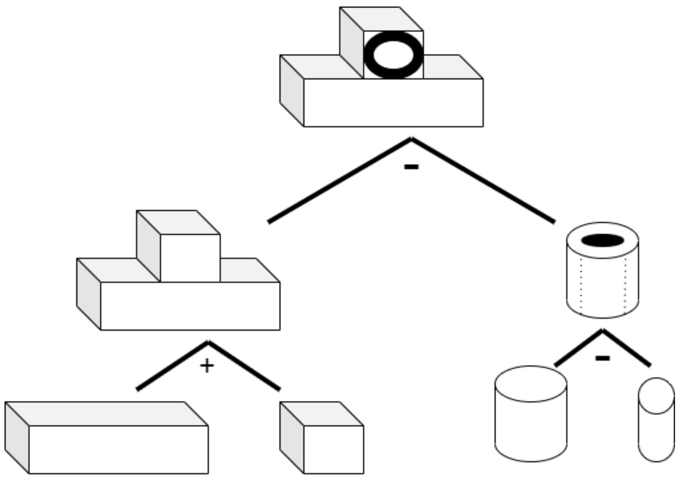


Figure 11: Example of a CSG tree

For every possible final object, where the number of leaves > 2, there are at least 2 different trees that produce the desired picture.

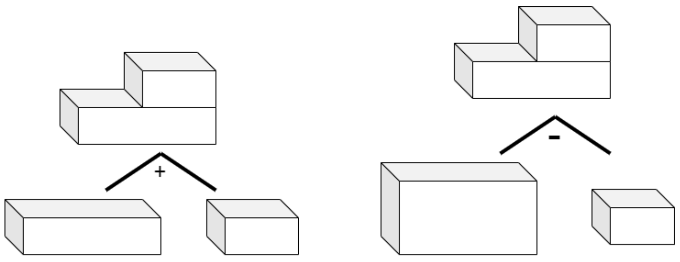


Figure 12: Different CSG trees may yield same output

For the purposes of 2D Graphics, we define primitives to be:

- circles $C^r_{x,y}$ with a center at a point (x, y) with a radius r
- triangles $T_{(x_1,y_1),(x_2,y_2),(x_3,y_3)}$ with vertices specified at (x_i, y_i)

And operators to be Boolean union $+$ and subtraction - .

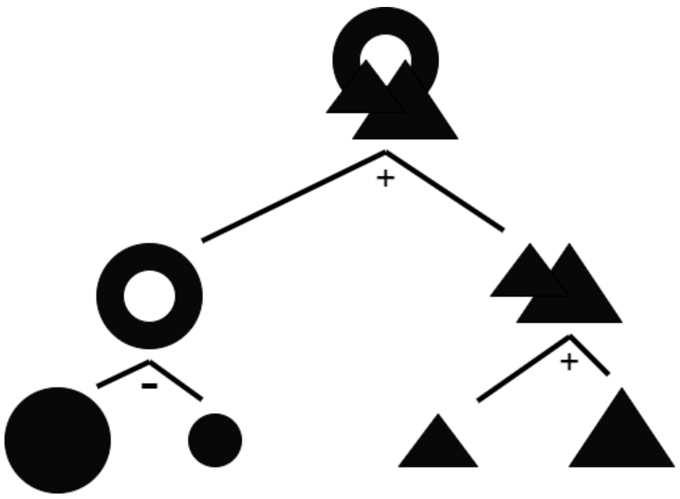


Figure 13: 2D CSG Tree

The CSG can be represented as context-free grammar:
$P -> P + P \mid P - P \mid C^r_{x,y} \mid T_{(x_1,y_1),(x_2,y_2),(x_3,y_3)}$

3.4 Read–Eval–Print Loop

A Read-Eval-Print Loop (REPL) is an interactive interpreter to a programming language. It helps bypassing the compile stage of the "code -> compile -> execute" cycle [7].

There are 4 parts to a REPL:

- A read function, which reads input from the keyboard

- An eval function, which evaluates code passed to it
- A print function, which formats and displays results
- A loop function, which runs the three previous commands until termination

REPL takes a CSG tree with predefined primitives and node operators to produce an output image. Such a process can be helpful when exposing of semantics is needed to be performed immediately.

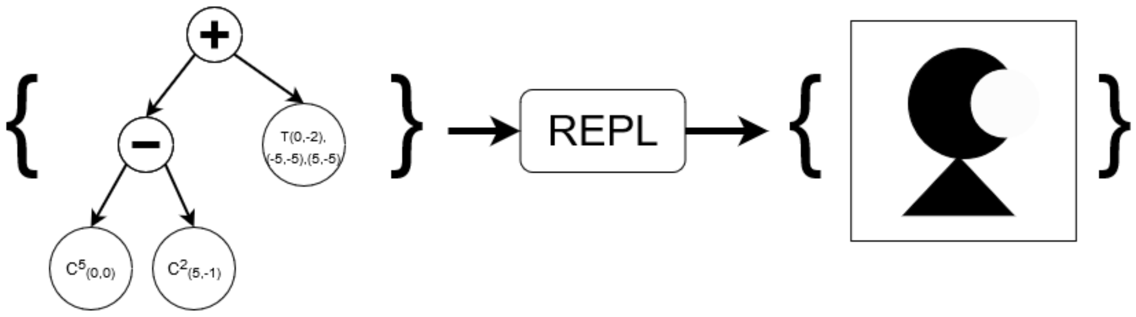


Figure 14: REPL renders a program into an image

In Figure 11, REPL executes one by one commands in the CSG tree which results into a final image. Such a tree can be described in code with any domain-specific language compliant with the REPL of choice. For example, program tree can have such a syntactic form, with the final product as the last command:

```
A = Circle(5,(0,0))
B = Circle(2,(5,-1))
C = A - B
D = Triangle((0,-2),(-5,-5),(5,-5))
E = C + D
```

4 Algorithm

4.1 Overview

Preceding sections have introduced a robust framework for optimization problems, the subsequent topic after that defined a structure for representation of 2D Graphics. Now, the focus shift towards synthesis of a sequence of commands that would yield the specification picture provided.

4.2 Representation of the data in the algorithm

4.2.1 Representation in the algorithm

Each chromosome would represent a sequence of commands (bits) $c_k = (c_1, .., c_k)$ of size k from the set of all commands C. In the case of CSG, bits are primitive types: circles and triangles with their respective parameters, and operators such as Boolean addition $+$ and subtraction - .Since our original CSG was a tree, we modify the primitives by adding one more parameter - color, which would transform the Boolean operators.

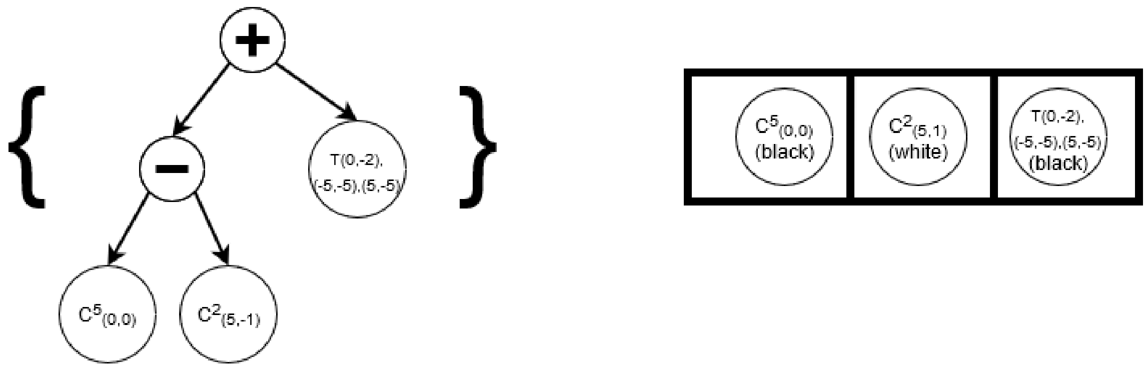


Figure 15: CSG as a tree and CSG as a chromosome

Figure 13 shows how a tree can be transformed from a tree into a sequence: by changing the color of the right subtree (child) in the subtraction operation node (in regards to the parent color) and then by printing out the leaves of the tree, left to right.

Now the tree becomes a sequence of commands where each cell represents a primitive that is to be displayed in the defined order (from left to right.)

4.3 Main loop of the algorithm

The loop of the algorithm is equal to the one presented in the Figure 2. The difference between the general algorithm and this implementation is in the steps such as Selection, Crossover and Mutation, which are presented below.

4.3.1 Initialization of initial population

While generating the population, three parameters are to be defined: length of a chromosome, population size, probability of selecting each command.

- length of a chromosome
- population size
- probability distribution of commands

Length of a chromosome defines the length of a sequence of commands in every program in the population. In the current implementation this length stays static throughout execution. The size of the respective population is the number of programs in the algorithm. The last parameter is the probability of selecting command c out of a pool of all commands.

4.3.2 Fitness function

In regards to the 2D graphics - each chromosome can be interpreted into an image (Figure 14) and then be compared to the specification with Intersection-over-Union method, which works well in practice.

Intersection over Union (IoU) is a well-known technique used for measuring the overlap between two bounding boxes. If the result $= 1$, then two pictures are identical, if result $= 0$, each pixel of the image is different between two images.

The formula for the IoU is $IoU = \frac{I}{U}$, where I is the number of identical pixels at their respective locations and U is the number of pixels in total.

```
image_1 <- First image
image_2 <- Second image

i = logical_and(image_1, image_2)
u = logical_or(image_1, image_2)

return i.sum() / u.sum()
```

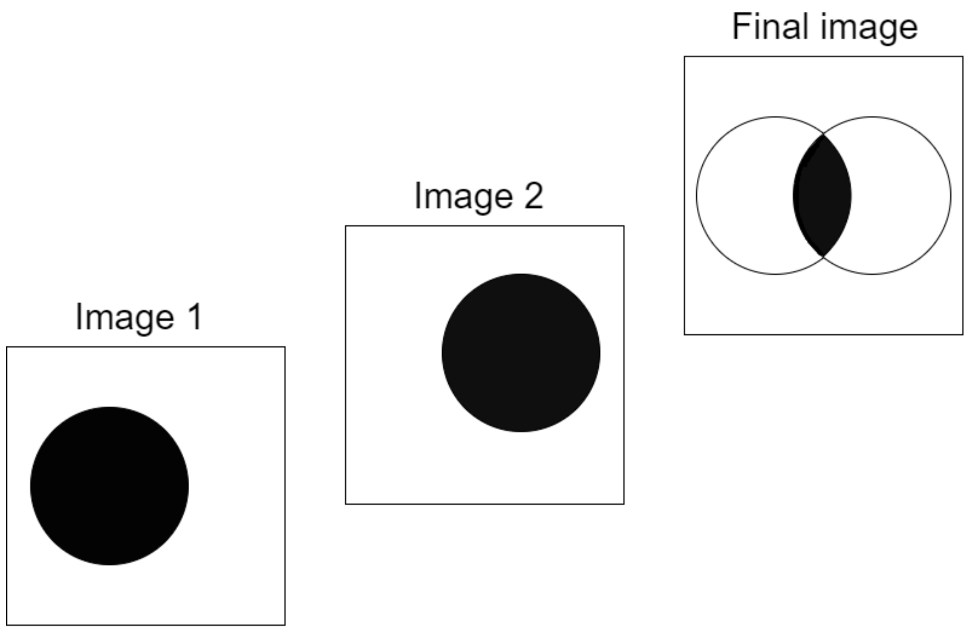


Figure 16: Example of IoU

In the example picture, we find the intersection between Picture 1 and Picture 2, which is represented on the Final Picture, and then calculate the ration of intersected pixels to the total number of pixels in the latter.

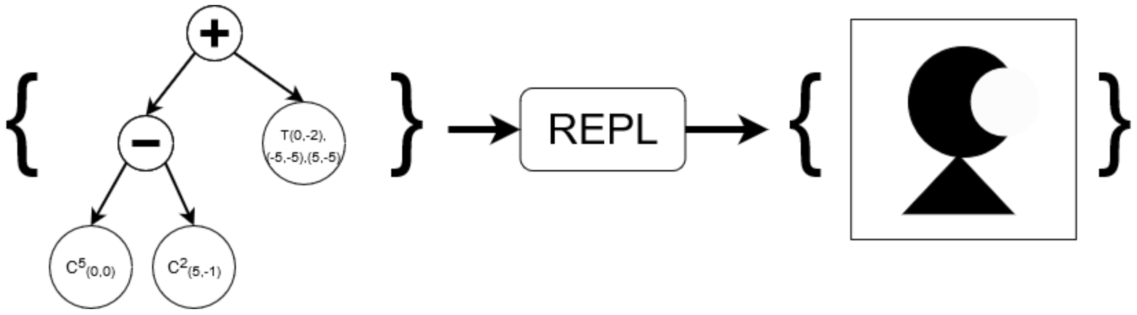


Figure 17: CSG as a tree and CSG as a chromosome

4.3.3 Selection

Selection is done by a tournament method - a random sample of 5 contestants is selected and the fittest gets to be a parent for the next stage. This has an advantage of being a ranking algorithm, rather than a fitness proportionate one, since a considerable amount of time at least two potential solutions have almost equal fitness score.

4.3.4 Crossover

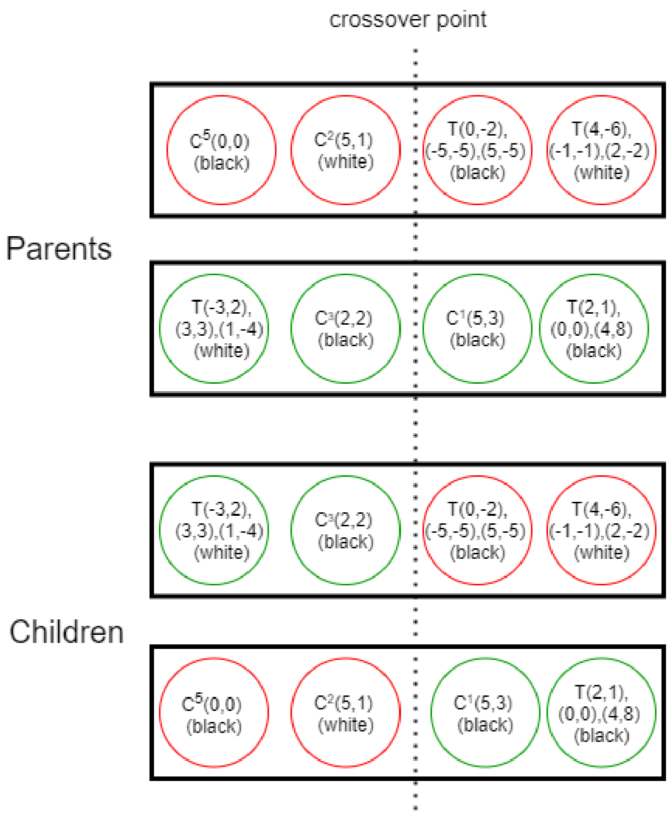
Current implementation is making use of single-point crossover, as described below:

A child c is created is follows:

- $c_k = p_{1_k}$, if $k < \frac{|p_1|+|p_2|}{4}$
- $c_k = p_{2_k}$, otherwise

where

- p_1 - first parent
- p_2 - second parent
- $|p_k|$ - length of a parent k



crossover point
Parents
C⁵(0,0) (black)
C²(5,1) (white)
T(0,-2), (-5,-5),(5,-5) (black)
T(4,-6), (-1,-1),(2,-2) (white)
T(-3,2), (3,3),(1,-4) (white)
C³(2,2) (black)
C¹(5,3) (black)
T(2,1), (0,0),(4,8) (black)
Children
T(-3,2), (3,3),(1,-4) (white)
C³(2,2) (black)
T(0,-2), (-5,-5),(5,-5) (black)
T(4,-6), (-1,-1),(2,-2) (white)
C⁵(0,0) (black)
C²(5,1) (white)
C¹(5,3) (black)
T(2,1), (0,0),(4,8) (black)

Figure 18: Crossover

4.3.5 Mutation

In our case, mutation can change not only the parameters of the command, but also the command itself.

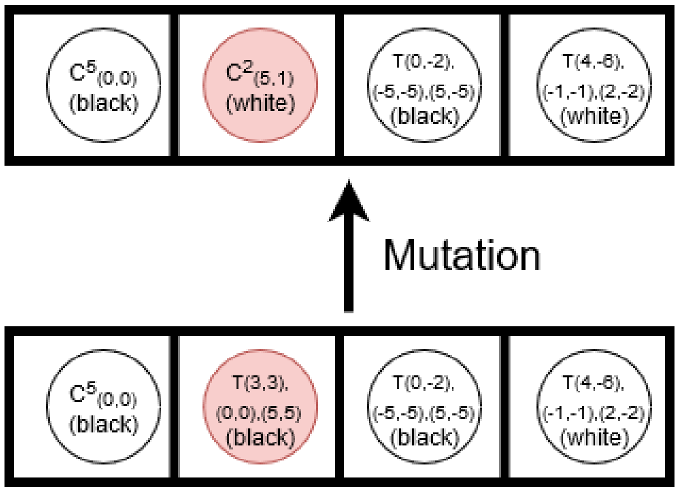

$C^5_{(0,0)}$ (black)
$C^2_{(5,1)}$ (white)
T(0,-2), (-5,-5),(5,-5) (black)
T(4,-6), (-1,-1),(2,-2) (white)
Mutation
$C^5_{(0,0)}$ (black)
T(3,3), (0,0),(5,5) (black)
T(0,-2), (-5,-5),(5,-5) (black)
T(4,-6), (-1,-1),(2,-2) (white)

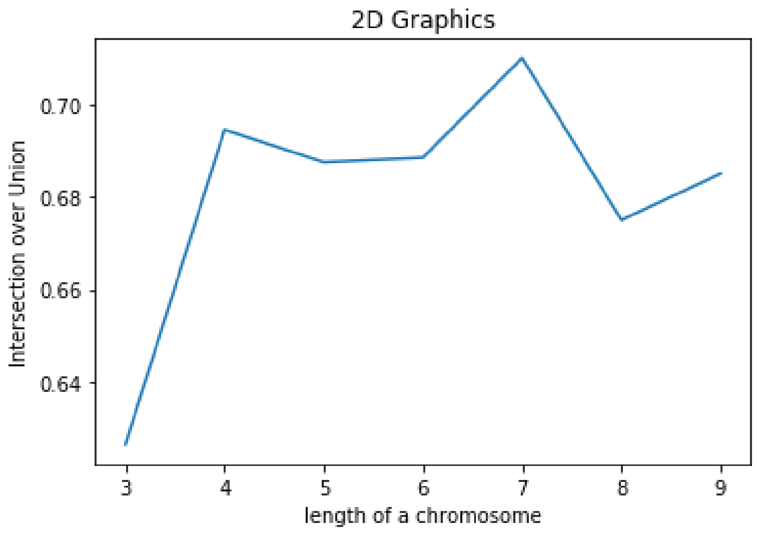
Figure 19: Mutation

5 Results

5.1 Length of a chromosome

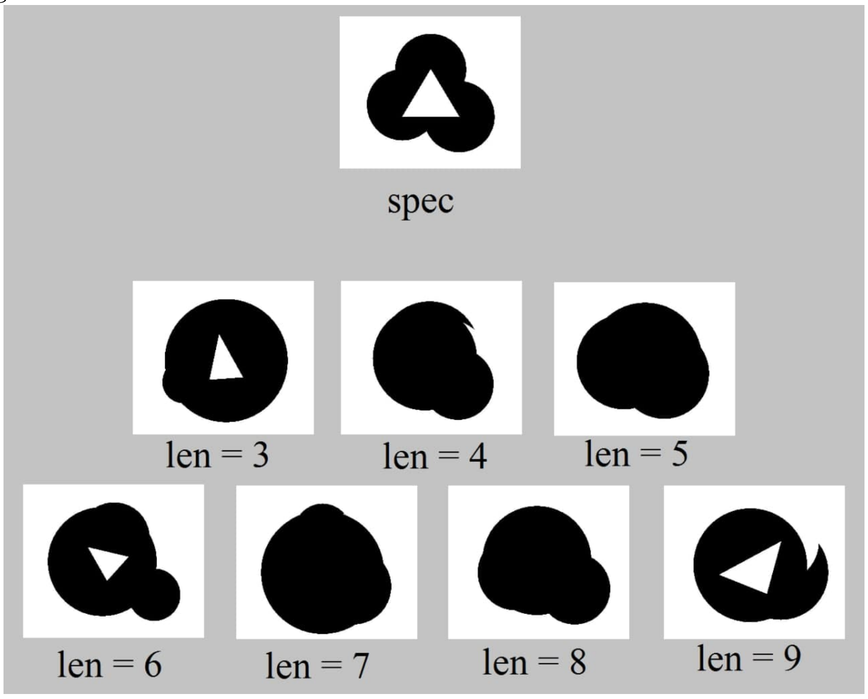
As it was discussed in 'Initialization of initial population', one of the parameters is the length of a chromosome, which symbolizes how many commands are there in a generated program.

Figure 20: Effects of length of a chromosome on the algorithm



As it can be seen in the graph above, the best length of a generated command spikes at 4 and 7 with a generally equal result for any other amount of commands. An example of how the number of commands changes the output in a picture form:

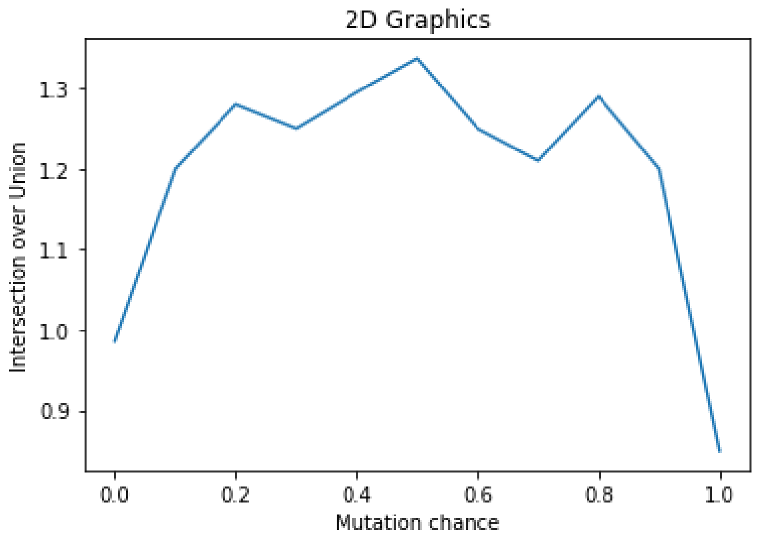
Figure 21: An example of reconstruction of a spec image, where len denotes length of a chromosome



5.2 Mutation

Another important parameter is a mutation percentage which informs about the chance that one particular command with its respective arguments would change after a crossover stage.

Figure 22: Effects of mutation on the algorithm

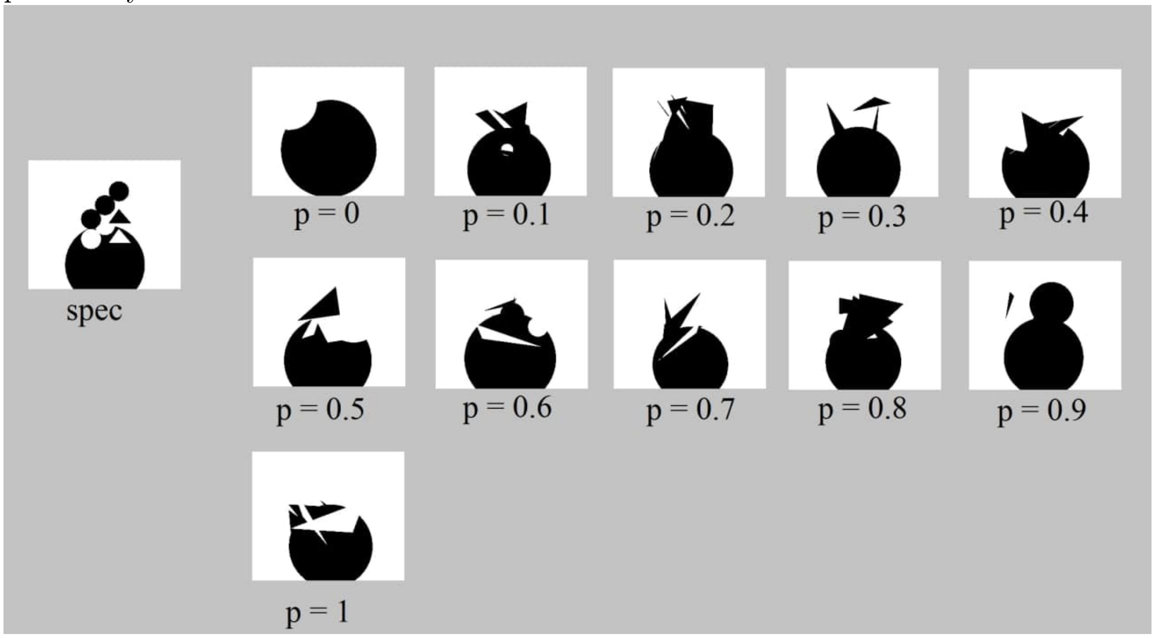


The best parameter for mutation probability turns out to be around 0.5, which will be used as a parameter for subsequent example generation. This plot echoes the idea that mutation rates of 0, 0.1 or 0.2 are poor decisions. [18] After these values it reaches its maximum at the point of 0.5 and after that the fitness score declines due to abundant amount of mutations.

This phenomena can be explained as: low values of mutation don't allow the heuristic to explore the possible search space enough, and as a result the local minimum is reached, which is suboptimal. High values of mutation probability, on the other hand, lead to random search and prevents the population to converge to any optimal solution.

Therefore a good balance is needed, where exploration and exploitation are at balance.

Figure 23: An example of reconstruction of a spec image, where p denotes probability of a mutation



5.3 Synthesis examples of 2D pictures

The parameters discussed above were used for this example generation. The picture from the specification is displayed on the top and the results are on the bottom. The spec programs consist of a various length of a chromosome ranging from 3 to 10. The used CSG spec programs' and resulting programs' code can be found in Appendix A.

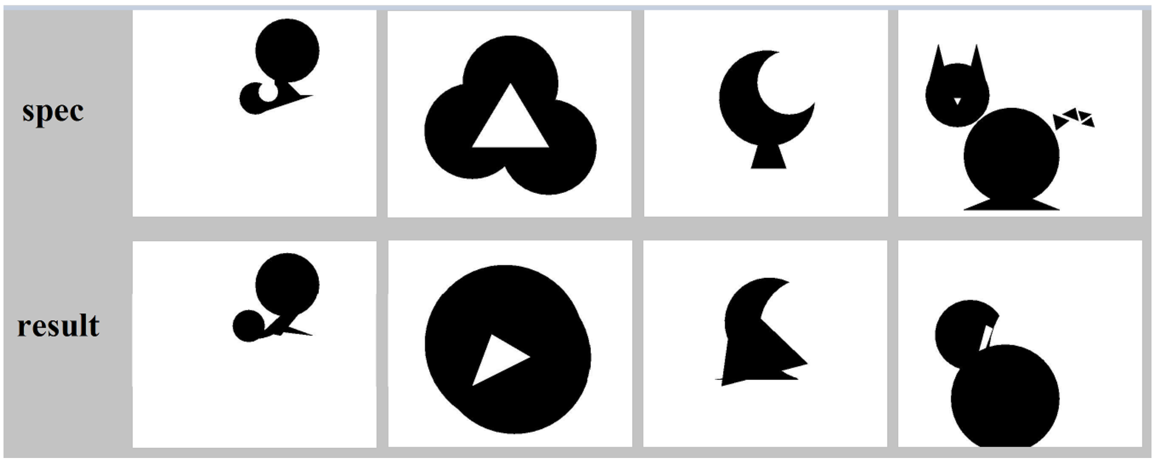


Figure 24: Generation results of 2D programs run through REPL

5.3.1 Generated commands

Section 4.3 showed the generated programs out of a specification. The code for the images number 2,3 and 4 are displayed from left to right, as it is shown in the Figure 21.

---

spec:

```
circle((-120,-50),150)
circle((0,100),150)
circle((120,-100),150)
triangle((0,100),(-120,-100),(120,-100), COLOR_WHITE)
```

generated code:

```
T((-130, -130),(190, -130),(110, -30),#000000)
T((-110, -130),(-30, -200),(-20, 140),#FFFFFF)
C((0, -180),70,#000000)
C((-20, 0),250,#000000)
T((150, -110),(-140, -80),(-180, 110),#000000)
C((10, -40),240,#000000)
T((-60, 30),(60, -40),(-120, -130),#FFFFFF)
```

---

spec:

```
circle((0,50),150)
circle((70,100),100, COLOR_WHITE)
triangle((0, 0),(60, -170),(-50, -170))
```

generated code:

```
T((130, -200),(70, -60),(-40, 100),#FFFFFF)
C((-110, -130),20,#FFFFFF)
C((10, 70),140,#000000)
T((100, -110),(70, -80),(-160, -110),#000000)
T((-10, 40),(-30, 10),(-140, -160),#FFFFFF)
C((140, 50),160,#FFFFFF)
T((-100, 160),(-140, -130),(130, -60),#000000)
```

---

spec:

```
circle((-200,60),100)
triangle((-300,60),(-260, 220),(-220, 60))
triangle((-180,60),(-140, 220),(-100, 60))
triangle((-210,50),(-200, 30),(-190, 50), COLOR_WHITE)
circle((-30,-130),150)
triangle((-180,-300),(-30, -240),(120, -300))

triangle((110,-50),(100, 0),(150, -20))
triangle((130,0),(170, 20),(180, -20))
triangle((180,10),(190, -20),(220, 0))
triangle((220,-10),(190, -30),(230, -40))
```

generated code:

```
T((150, -110),(-120, -200),(-40, 80),#000000)
T((80, -180),(-20, 160),(-160, -170),#FFFFFF)
T((-190, 140),(140, 110),(-30, 50),#000000)
T((80, -180),(-140, -170),(20, 120),#FFFFFF)
T((20, -100),(-10, -50),(50, 0),#FFFFFF)
C((-50, -10),280,#FFFFFF)
T((-170, 110),(80, -60),(90, 20),#FFFFFF)
T((-120, -190),(170, -70),(-110, 130),#000000)
C((50, 20),210,#FFFFFF)
T((-180, -170),(70, -150),(-40, 190),#FFFFFF)
T((90, -70),(0, 160),(130, 0),#FFFFFF)
C((100, -40),200,#FFFFFF)
C((-160, 30),110,#000000)
T((20, -130),(-80, -190),(0, 110),#FFFFFF)
T((-90, 50),(-110, 60),(-180, -200),#FFFFFF)
T((-100, -20),(-150, -120),(60, -60),#000000)
C((50, 100),40,#000000)
T((180, 60),(150, -60),(100, 60),#000000)
C((140, -30),240,#FFFFFF)
C((-50, -170),170,#000000)
```

6 Limitations

- The method presented relies heavily on a complete and correct specification, which oftentimes is unavailable, difficult to write or is constantly changing.
- Unlike Neural Network approaches, evolutionary algorithms do not offer re-usability. For every new specification a new run is required.
- For every new specification different set of parameters can be optimal, which results in a higher number of runs needed for meaningful results to appear.

7 Related Work

This thesis was motivated by a recent paper combining REPL and RL approaches [14]. Currently, the general trend for solving such problems is to use execution-guided neural program synthesis, though heuristics are still viable [21] [6].

8 References

[1] John Daniel Bagley. The behavior of adaptive systems which employ genetic and correlation algorithms : technical report. 1967.

[2] J. E. Baker. Reducing bias and inefficienry in the selection algorithm. In *ICGA*, 1987.

[3] Shumeet Baluja and Rich Caruana. Removing the genetics from the standard genetic algorithm. Technical report, USA, 1995.

[4] H. Bremermann. Optimization through evolution and recombination. 1962.

[5] Rogson M Bremermann H J and Salaff S. Search by evolution. pages 157 – 167, 1965.

[6] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019.

[7] Tom Collins. `https://stackoverflow.com/a/13612824`.

[8] Computer Programming. Computer programming — Wikipedia, the free encyclopedia, 2020.

[9] Charles Darwin. *The origin of species.* Everyman's library. Dent, 1936.

[10] Lawrence Davis. *Genetic Algorithms and Simulated Annealing.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[11] Kenneth De Jong, David Fogel, and Hans-Paul Schwefel. *A history of evolutionary computation*, pages A2.3:1–12. 01 1997.

[12] Eduemoni. Cad model of a mouse. `https://en.wikipedia.org/wiki/User:Eduemoni`.

[13] A. E. Eiben and James E. Smith. *Introduction to Evolutionary Computing.* Natural Computing Series. Springer, 2003.

[14] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, Execute, Assess: Program Synthesis with a REPL. *arXiv e-prints*, page arXiv:1906.04604, June 2019.

[15] J.D. Foley, F.D. Van, A. Van Dam, S.K. Feiner, J.F. Hughes, E. Angel, and J. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley systems programming series. Addison-Wesley, 1996.

[16] R. M. Friedberg. A learning machine: Part i. 2(1), 1958.

[17] D. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In *FOGA*, 1990.

[18] R.N. Greenwell, J.E. Angus, and M. Finck. Optimal mutation probability for genetic algorithms. *Mathematical and Computer Modelling*, 21(8):1 – 11, 1995.

[19] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992.

[20] K.A. De Jong. *An analysis of the behavior of a class of genetic adaptive systems*. PhD thesis, University of Michigan, Ann Arbor, MI, USA, 1975.

[21] Krzysztof Krawiec. Heuristic approaches to program synthesis : Genetic programming and beyond. `http://phdopen.mimuw.edu.pl/zima14/krawiec-slides.pdf`.

[22] Sean Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[23] Vijini Mallawaarachchi. How to define a fitness function in a genetic algorithm? `https://towardsdatascience.com/how-to-define-a-fitness-function-in-a-genetic-algorithm-be572b9ea3b4`.

[24] Terrence Masson. The computer graphics book of knowledge. `https://www.cs.cmu.edu/~ph/nyit/masson/history.htm`.

[25] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

[26] Anna Mleczko. How many developers are there in the world in 2020? `https://future-processing.com/blog/how-many-developers-are-there-in-the-world-in-2019/`.

[27] Gil Press. Ai and automation by the numbers: Predictions, perceptions, and proposals. `https://forbes.com/sites/gilpress/2017/03/30/ai-and-automation-by-the-numbers-predictions-perceptions-and-proposals/?sh=262d512a2bb3`.

[28] M.M.M. SARCAR, K.M. RAO, and K.L. NARAYAN. *Computer Aided Design and Manufacturing*. PHI Learning, 2008.

[29] Jayshree Sarma and Kenneth De Jong. C2.7 generation gap methods. 12 2003.

[30] Pradnya Vikhar. Evolutionary algorithms: A critical review and its future prospects. pages 261–265, 12 2016.

[31] Joe Warren. Why is computer graphics important? `https://www.cs.rice.edu/~jwarren/360/outline/subsection3_1_2.html`.

[32] Saneh Yadav and Asha Sohal. Comparative study of different selection techniques in genetic algorithm. *International Journal of Engineering Science*, 07 2017.

Program Code

January 28, 2021

[1]:
```
import turtle
import io
import numpy as np
import math

from random import random, randint, seed
from statistics import mean
from copy import deepcopy
from sys import platform
from PIL import Image
```

[2]:
```
COLOR_WHITE='#FFFFFF'
COLOR_BLACK='#000000'
POP_SIZE        = 100        # population size
DEPTH           = 8          # maximal initial random tree depth
GENERATIONS     = 120        # maximal number of generations to run evolution
TOURNAMENT_SIZE = 5          # size of tournament for tournament selection
XO_RATE         = 0.8        # crossover rate
PROB_MUTATION   = 0.2        # per-node mutation probability
GOAL_IMAGE_NAME = "goal"     # name of the specification image
SAVE_PATH       = "pictures/" # path for saving the generated images
```

The following commands represent the primitives for the REPL, such as Cirles and Triangles. The REPL of choice is Python Turtle.

[4]:
```
def circle(coord, radius, color='#000000'):
    t.goto(coord[0], coord[1] - radius)
    t.fillcolor(color)
    t.color(color)

    t.down()
    t.begin_fill()
    t.circle(radius)
    t.end_fill()
    t.up()


def triangle(
    coord1,
    coord2,
    coord3,
    color='#000000',
    ):
    t.goto(coord1[0], coord1[1])
    t.fillcolor(color)
    t.color(color)

    t.down()
    t.begin_fill()
    t.goto(coord2[0], coord2[1])
    t.goto(coord3[0], coord3[1])
    t.goto(coord1[0], coord1[1])
    t.end_fill()
    t.up()
```

```
[5]: FUNCTIONS = [circle, triangle]
     TERMINAL_COLORS = [COLOR_WHITE, COLOR_BLACK]
     TERMINAL_RADIUSES = list(range(0, 410, 10))
     TERMINAL_POINTS = [(x, y) for x in list(range(-200, 200, 10)) for y in
                        list(range(-200, 200, 10))]
```

The save function which takes a screenshot of the REPL output.

```
[6]: def save(name=GOAL_IMAGE_NAME):
         ps = turtle.getscreen().getcanvas().postscript(colormode='color')
         img = Image.open(io.BytesIO(ps.encode('utf-8')))
         img.save(SAVE_PATH + name + '.jpg')
```

The implementation of the fitness function of choice: Intersection-over-Union.

```
[7]: def intersection_over_union(img1, img2='test'):
         image1 = np.asarray(Image.open(SAVE_PATH + img1 + '.jpg'))
         image2 = np.asarray(Image.open(SAVE_PATH + img2 + '.jpg'))
         image_empty = np.asarray(Image.open(save_path + 'empty.jpg'))

         image1_i = np.logical_not(np.logical_and(image1, image_empty))
         image2_i = np.logical_not(np.logical_and(image2, image_empty))

         i = np.logical_and(image1_i, image2_i)
         u = np.logical_or(image1_i, image2_i)

         return i.sum() / u.sum()
```

An example of generating a specification image through primitive commands.

[ ]:
```
try:
    t = turtle.Turtle()
except:
    t = turtle.Turtle()
wn = turtle.Screen()
t.up()
t.hideturtle()
turtle.tracer(0, 0)

circle((0, -200), 200)
circle((0, 0), 50, COLOR_WHITE)
circle((-70, -70), 50, COLOR_WHITE)
triangle((70, -20), (130, -90), (20, -90), COLOR_WHITE)

circle((70, 170), 50)
circle((0, 100), 50)
circle((-70, 30), 50)
triangle((70, 80), (130, 10), (20, 10))

save('goal')

turtle.bye()
```

The EAQueue class is the representation of a chromosome in a population, which is a queue of commands (primitives). Each chromosome can be transformed into a resulting image through REPL.

[ ]:
```
class EAQueue:

    def __init__(self):
        self.queue = []

    def __len__(self):
        return len(self.queue)

    def print_queue(self):
        print '{'
        for i in range(len(self.queue)):
            self.queue[i].print_node()
        print '}'

    def random_queue(self, max_depth):
        for _ in range(max_depth):
            self.queue.append(self.random_node())

    def random_node(self):
        if random() > 0.5:
            return CircleNode.get_random_circle()
        else:
            return TriangleNode.get_random_triangle()

    def mutation(self):
        for i in range(len(self.queue)):
            if random() < PROB_MUTATION:
                self.queue[i] = self.random_node()

    def crossover(self, other):
        if random() < XO_RATE:
            if len(other) < len(self.queue):
                self.queue[math.floor(len(other) / 2):len(other)] = \
                    other.queue[math.floor(len(other) / 2):]
            else:
                self.queue = self.queue[0:math.floor(len(self.queue)
                        / 2)] + other.queue[math.floor(len(self.queue)
                        / 2):]

    def compute_queue(self):
        for i in range(len(self.queue)):
            self.queue[i].compute()
```

CircleNode and TriangleNode are primitives which are stored inside the EAQueue.

[ ]:
```
class CircleNode:

    def __init__(
        self,
        center,
        radius,
        color,
        ):
        self.command = circle
        self.center = center
        self.radius = radius
        self.color = color

    def compute(self):
        return self.command(self.center, self.radius, self.color)

    def print_node(self):
        print 'C(' + str(self.center) + ',' + str(self.radius) + ',' \
            + self.color + ')'

    @staticmethod
    def get_random_circle():
        return CircleNode(TERMINAL_POINTS[randint(0,
                          len(TERMINAL_POINTS) - 1)],
                          TERMINAL_RADIUSES[randint(0,
                          len(TERMINAL_RADIUSES) - 1)],
                          TERMINAL_COLORS[randint(0,
                          len(TERMINAL_COLORS) - 1)])


class TriangleNode:

    def __init__(
        self,
        first_point,
        second_point,
        third_point,
        color,
        ):
        self.command = triangle
        self.first_point = first_point
        self.second_point = second_point
        self.third_point = third_point
        self.color = color

    def compute(self):
        return self.command(self.first_point, self.second_point,
                            self.third_point, self.color)

    def print_node(self):
        print 'T(' + str(self.first_point) + ',' \
            + str(self.second_point) + ',' + str(self.third_point) \
            + ',' + self.color + ')'

    @staticmethod
    def get_random_triangle():
        return TriangleNode(TERMINAL_POINTS[randint(0,
                            len(TERMINAL_POINTS) - 1)],
                            TERMINAL_POINTS[randint(0,
                            len(TERMINAL_POINTS) - 1)],
                            TERMINAL_POINTS[randint(0,
                            len(TERMINAL_POINTS) - 1)],
                            TERMINAL_COLORS[randint(0,
                            len(TERMINAL_COLORS) - 1)])
```

Helper functions, which correspond to some stages of Evolutionary Algorithm mainloop, such as: initialization of the population, calculation of the fitness function and selection.

```
[ ]: def init_population():
         pop = []
```

```
    for _ in range(POP_SIZE):
        q = EAQueue()
        q.random_queue(MAX_DEPTH)
        pop.append(q)

    return pop


def fitness(individual):
    individual.compute_queue()
    save('candidate_solution')

    return intersection_over_union('candidate_solution',
                                   GOAL_IMAGE_NAME)


def save_best_run(individual):
    individual.compute_queue()
    save('individual')


def selection(population, fitnesses):
    tournament = [randint(0, len(population) - 1) for i in
                  range(TOURNAMENT_SIZE)]
    tournament_fitnesses = [fitnesses[tournament[i]] for i in
                            range(TOURNAMENT_SIZE)]

    return deepcopy(population[tournament[tournament_fitnesses.
↪index(max(tournament_fitnesses))]])
```

The mainloop of the Evolutionary Algorithm, as presented in the Figure 2 of the thesis, which saves a final approximated image in a folder specified in constant values.

[ ]:
```
# Initialization of REPL

try:
    t = turtle.Turtle()
except:
    t = turtle.Turtle()
wn = turtle.Screen()
t.up()
t.hideturtle()
turtle.tracer(0, 0)

best_of_run = None
best_of_run_f = 0
best_of_run_gen = 0
```

```python
# Initialize population

population = init_population()

# Fitness calculation

fitnesses = [fitness(population[i], GOAL_IMAGE_NAME) for i in
             range(POP_SIZE)]

for gen in range(GENERATIONS):
    nextgen_population = []
    for i in range(POP_SIZE):

       # Selection

        parent1 = selection(population, fitnesses)
        parent2 = selection(population, fitnesses)

       # Crossover

        parent1.crossover(parent2)

       # Mutation

        parent1.mutation()
        nextgen_population.append(parent1)

    population = nextgen_population
    fitnesses = [fitness(population[i], GOAL_IMAGE_NAME) for i in
                 range(POP_SIZE)]

    if max(fitnesses) > best_of_run_f:
        best_of_run_f = max(fitnesses)
        best_of_run_gen = gen
        best_of_run = \
            deepcopy(population[fitnesses.index(max(fitnesses))])

save_best_run(best_of_run)
```