Evolving a Search Procedure for Program Synthesis

Michał Okoń
Supervisor: Sebastijan Dumančić
EEMCS, Delft University of Technology, The Netherlands

23-6-2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

In recent months, researchers developed several new search procedures to augment the process of program synthesis. While many of them performed better than their predecessors, the proposed solutions are still far from ideal. One possible way of overcoming the shortcomings of single search methods is employing genetic algorithms, which have been proven useful in many tasks of similar scale. This paper aims to answer the question of whether it is possible to utilize that sort of algorithms to find an efficient combination of search procedures in a program synthesis problem. An implementation of a genetic algorithm is proposed with parameters and operators chosen through a literature study and a series of experiments on three different domains. To outline different approaches to program synthesis, two fitness functions are examined. Finally, evolved search procedures are discussed and compared with the already existing solutions. One of them in particular, namely a combination of Brute and A* algorithms, manages to surpass their singular counterparts in certain cases.

1 Introduction

The problem of program synthesis is often described as a task that can completely revolutionize the field of Artificial Intelligence and Computer Science in general [1]. The ability to generate programs from examples could greatly reduce the amount of work spent on engineering new software and help improve the already existing solutions. Extensive research has been performed on how programs can be generated. The majority of the procedures can be reduced to defining the space of the program (most often as a domain-specific language) and using a variety of search techniques to find correct and efficient programs[2]. One of the recent advancements in the area, was the development of Brute [3]- an ILP (inductive Logic Programming [4]) system that invents library predicates using brute-force methods and, consequently, uses best-first search with an example-dependent loss function to build programs. Since the loss function may take different values depending on how close our current outputs are to the desired ones, Brute outperforms many already existing ILP systems. However, there is still plenty of possibilities for improvement and promising enhancements to this procedure have been recently investigated.

In the last research project performed at TU Delft, students examined a variety of different search procedures to efficiently search through the program space containing allowed programming expressions. The main goal was to improve the search procedures used by Brute. Each of the algorithms was tested on three domains proposed by the creators of Brute[3]- robot path planning, ASCII (pixel) art, and string transformations. The researched algorithms were A*[5], Very Large Neighborhood Search with Variable-Depth Neighborhood Search (Vlute)[6], Metropolis-Hastings[7] and Genetic Algorithm (VanillaGP)[8].

From the obtained results, it is clear that the performance of most methods is very domain-dependent and, even though substantial improvements have been made, the algorithms often fail to provide satisfactory solutions. Each of the developed algorithms achieves a different ratio between exploration (seeking the solution in the further unexplored areas) and exploitation (searching the neighborhood of promising regions), leading to performance varying from task to task. Finding a method to eliminate part of the drawbacks, while maintaining their positive traits, could lead to a great increase in both the quality of found programs and the search speed.

One intriguing way to combine the singular powers of those search procedures is through the means of genetic algorithms. This subclass of evolution algorithms mimics the evolutionary processes observed in nature. Starting from an initial population of individuals, the algorithm advances from one generation to another trying to perpetuate the most desirable traits until an acceptable solution is found [9]. This family of algorithms has found application in solving many problems characterized by difficulty with finding solutions using more tailored methods. The fields in which the Genetic Algorithms have proven to be useful include Machine Learning, Combinatorial Optimisation, and Image Processing [10]. Having discovered an efficient way to encode a combination of search procedures as a chromosome, one could, with help of appropriate metrics, evaluate which sequences of methods show the most promise and propagate their qualities into the next generations using a range of genetic operators.

This leads to the formulation of the main research question:

Q: Is it possible to develop an efficient search procedure as a combination of simpler procedures of various properties by means of genetic algorithms?

The main research question can be divided into several subquestions, which correspond to the sequence of measures I will have to take to construct an adequate Genetic Algorithm.

- **Q:** How can we construct a genotype of a single specimen (chain of search procedures constituting one combined search procedure)?
- **Q:** How do we evaluate the obtained search procedures? In other words, how is fitness calculated?
- **Q:** How do we select the fittest individuals?
- **Q:** What genetic operators, and with what probability, do we use to gradually improve the performance of the program synthesizer?

Due to the inherent limitations of this research, the paper mostly targets selecting two factors that seemed the most ambiguous and influential- the mutation probability and fitness functions, for different values of which the results are verified experimentally. The rest of the parameters are picked through the literature study.

In the following sections, I try to address each of the posed issues. In the section 2, I summarize the premises of Program Synthesis and Genetic Algorithms- the two main building blocks of this paper. Next, the details of the created solution, as well as the justification for the made choices, are outlined in the section 3. The setup of the experiments and

the discussion about the obtained results can be found in section 4 with the concluding remarks and the perspective of future work on this topic located in section 5. Lastly, section 6 contains the remarks about the responsible research.

2 Related Work

2.1 Program Synthesis

As mentioned in the introduction, Program Synthesis is a process of finding a program in a specific programming language to meet specifications representing the user intent [1]. The user intent can take various forms such as natural-language descriptions or logical specifications. In our case, the specifications are the sets of input and output such that for each input, the desired output that the synthesized program should return is given. A variety of heuristic loss functions (also called distance functions) is employed to assess how close the output returned by the synthesized programs lies to the desired solutions.

The programming language comprises the search space. When choosing the programming language to use, we should aim to find a set of expressions that are both expressive (i.e. working program can be constructed) and efficient (i.e. the search space is small enough to support fast search procedures). The solution described in this paper makes use of sets of complex tokens, which are invented from more primitive expressions building the Domain Specific Language (DSL). Examples of those primitive expressions are functions such as [MoveUp] or [DropBall] in the robot path planning domain.

To search the program space, a range of search procedures is used. To fully present the scope and foundations of the research, it is crucial to summarize the main search algorithms, which will be the building blocks of evolved search chains. The following subsections look at Brute and four new techniques developed by the students in the course of the past research project. Every technique shows a different level of promise in each of the considered domains.

All the discussed algorithms consist of invent and search phases. In the invent stage, complex tokens building the programs are invented from a set of primitive expressions. In the original papers describing the implementation of the algorithms, every one of them had a separate invent stage tailored to the needs of the algorithm. While efficient in some cases, this method did not support valid comparisons between search procedures since the choice of invent methods had a huge impact on the performance of search algorithms. In the recent changes made to the code, the invent methods have been made equal for each algorithm. Now, a static invent is used, which invents thousands of tokens by pairing them or simply utilizing single expressions along with control tokens such as 'if' statements. On an identical account, loss functions are shared across the algorithms.

Brute

Brute starts building a program that aims to satisfy the given specifications (input-output pairs) by extending an empty hypothesis with new tokens [3]. To support the best-first search, a priority queue is created based on the values returned by the loss function; better-performing programs are put higher in the polling priority, hence the greedy nature of the algorithm. For each new program, every invented token is appended to its end and the resulting output is compared to the desired one using the loss function. The procedure continues until a satisfactory program is found. In terms of performance, Brute surpasses the Metagol- another very efficient program synthesis system. One major deficiency of Brute is the tendency for converging in local minima, especially in the larger domains.

A*

A procedure that utilizes a heuristic-based search to alleviate the greedy nature of Brute [5]. The search algorithm operates on a graph where nodes are states (e.g. state of the board in the robot path planning) and the edges are tokens that lead to the specific state. In addition to the domain-specific heuristic loss function, the number of steps taken to current states is also considered in the node cost calculation. The cost of a single path is treated as the cost of a token assigned to the edge. This leads to favoring more concise programs and mitigates the problem of being stuck in local minima. Those improvements result in a drastic growth of performance in the part of the domains.

Very Large Neighborhood Search with Variable-Depth Neighborhood Search (Vlute)

When searching through the program space, each potential program is assigned a depth depending on how many transition and control tokens are used to construct it [6]. At each stage, the algorithm keeps track of the best-found solution (gauged by the loss function) and tries to destroy and rebuild potential programs by removing and inserting a part of the building tokens. Depending on the parameters, if a better solution has not been found for long enough, the allowed maximum depth is increased, consequently enlarging the search space.

Metropolis-Hastings

A Metropolis-Hastings-based algorithm is constructed with the stationery distribution dependent on the domain-specific cost functions and each of the mutations is assigned a probability of occurring [7]. On each iteration, a change in the structure of the code might transpire. The better the program obtained from performing such mutation compared to the previous program, the higher the probability of transition. Because of its random nature, this algorithm manages to outperform other algorithms in the robot path-planning domain.

VanillaGP

A direct and very simplistic genetic programming approach to the problem of program synthesis. The initial population is generated as a set of single tokens [8]. Consequently, individuals for breeding are selected based on a roulette wheel selection procedure [10] with fitness calculated as a reverse of the domain-specific cost function. Mutations include operations such as removing or adding new tokens. One should not confuse this genetic programming process with the one developed in this paper since they are unrelated.

2.2 Genetic Algorithms

The main premise of the genetic algorithms is not hard to comprehend since they resemble the processes that we ob-

serve in nature. With the problem encoded as a set of chromosomes, we move from one generation of individuals to another while trying to preserve traits that are the most favorable [9]. The following subsection will focus on a general description of this process, while the details regarding the case-specific implementation of the genetic algorithm are described in the methodology.

Coming up with the proper coding of a problem is the first and perhaps most crucial step of a genetic algorithm. Chosen encoding directly impacts other stages of the process such as the kind of genetic operators that can be employed. The problems need to be represented as a set of parameters (called genes), which in turn are structured in the sequences called chromosomes [10]. The chromosome has usually been described as binary strings so that the cardinality of used symbols is minimal. However, using floating-point representation is also fairly common.

With the properly encoded problem, the initial generation of chromosomes is populated with random individuals. In each generation, the fittest individuals are selected by first calculating their fitness - value signifying how close to the desired solution, a given individual resides and then selecting the ones who get the chance to pass on their genes. The fitness calculation method is highly problem-specific [9], therefore, the details of my implementation are described in the methodology section.

The next step is to select the individuals to reproduce. While choosing the selection procedure, I have considered the three most common approaches: Proportional Roulette Wheel Selection, Rank-Based Roulette Wheel Selection, and Tournament Selection. Since incorporating genetic programming in finding a search procedure for program synthesis is a novel and unexplored problem, the final decision was based on the experiments performed in the domain of TSP (Travelling Salesman Problem) [11]. In the paper, it was discovered, that the Tournament Selection strategy manages to discover efficient solutions with a smaller number of generations and faster execution time of one iteration than two other strategies. However, this only applies to small populations. Since the sizes of the populations employed in the program synthesis problem are rather limited, Tournament Selection has been chosen. The way the Tournament Selection strategy functions is simple. First, some number of individuals is randomly selected from the population. Next, a specimen with the highest fitness is chosen from this pool to become one of the parents of the future generation [11].

After the selection, a range of mutation operators is used to recombine and mutate the selected parents. First, the crossover operators are applied with a certain probability, which is usually situated between 0.6 and 1.0 [12], to the pairs of parents. The crossover operators combine the genotype of two parents to create a new individual sharing part of their traits. If the crossover does not occur, parents simply move on to the next generation in unchanged forms. After the recombination stage, each resulting individual has a predetermined chance of being subjected to mutation by applying modifications to one or more of the genes. The probability of mutation is usually set between 0.01 and 0.05, however, the literature tends to sharply disagree in that matter[12], which is why it is encouraged to elect those values experimentally.

The whole process described above is repeated until a given number of iterations has passed or the desired solution is found. Since it is hard to define a point in the evolution process, when we are fully satisfied with the outcome, the generation limit is established experimentally and declared as a point after which no significant progress is observed.

3 Methodology

The following section describes the process of constructing a genetic algorithm suited for finding a search procedure used in program synthesis. It looks into methods of defining a chromosome, generating the initial population, calculating fitness, selecting the best individuals, performing mutation and crossover operations, and evaluating the final program. The procedure described here is later verified experimentally.

3.1 Defining a chromosome

Even though chromosomes have traditionally been defined as binary strings, nothing stands in the way of using other forms of representation, such as more meaningful non-numeric or floating-point ones. That way, one can access a vast range of real-number operators, discussed in the coming sections, that are more problem-specific and easier to reason about [13].

With that in mind, two versions of problem encoding have been considered. Both of them are expressed as an ordered list of pairs with each pair consisting of a search procedure and a number. Consecutive procedures begin searching the program space from the point their predecessor left off. The difference between those two methods lies in what the number represents. In the first considered version, the number was an integer corresponding to the number of iterations each algorithm is executed for. Unfortunately, due to the lack of a consistent definition of what an iteration is and varying execution times of a single iteration in the existing codebase, this method turned out to be highly inefficient. In that case, generating a balanced initial population was extremely difficult. The second version alters the definition of the number in a pair. Now, the number is a float depicting the maximum execution time per task (timeout) instead of the iterations. Since time is a metric that remains unambiguous across different search methods and domains, one can easily investigate the whole process and employ meaningful mutation operators. Unfortunately, this method also makes the solution constrained to the system the experiments were performed on, as the computation speed differs across different computers. To alleviate this issue, the specifications of the used computing cluster are presented in the appendix .1. An example of the final chromosome can be seen in figure 1.

Figure 1: Example of a chromosome represented as a chain of search procedures paired with their timeouts. First, Brute is executed for 0.3 seconds, then A* for 0.4 seconds and MCTS for 0.5 seconds.

3.2 Generating the initial population

Having defined the chromosome encoding, the next task is to invent a method of generating the initial population. Several main rules need to be kept in mind. Firstly, each search procedure should be given an equal chance regarding the execution time and frequency of appearance. That is where our way of encoding the problem proves useful. Since each search procedure is assigned an execution time, fairness can be achieved by randomly selecting a search procedure with an equal probability each and assigning the execution time from a uniform distribution that is identical for every search.

Extremely long search procedures do not only add unnecessary complexity but also obscure the understanding of the resulting search chains. The second important rule is to ensure that the chromosomes do not grow to unreasonable sizes. This can be accomplished by both implementing safe genetic operators, which will be discussed later, and introducing appropriate constraints to the initial populations. One simple constraint, that deals with the posed issue, is implementing an upper boundary on the size of a chromosome. Before search procedures are sampled, the size of a chromosome is determined by selecting a random number from the range of 1 to the maximum size. In the early experiments, the maximum sequence size of around 5 appears to give the most valuable results.

One last parameter that needs to be fixed is the size of the population of a single generation, which should be high enough to explore distant areas of the search space and not become stagnated in local minima. At the same time, performing a full program synthesis on hundreds of tasks is very computationally expensive, therefore, I need to strike a balance between a meaningful variety of the genome and feasible computation cost. The population of size between 20 and 40 seems to fit into both of those categories with an exception of the robot path planning domain which allows employing as high as 100 individuals per generation.

3.3 Calculating fitness

With the initial population generated, one needs to construct a way to evaluate the specific individuals. The evaluation should be based on the outcome of a search synthesizer employing the search procedure of the specimen under evaluation. To perform training and evaluation, examples derived from three different domains have been used, which have already been extensively utilized for experiments by other researchers in the field of program synthesis [3]. Those domains are: robot path planning, string transformations, and ASCII art. For each task, consisting of one or more examples, a separate program is found. To assess how close the output given by the program is to the correct solution, example-based domain-specific loss functions are employed (e.g. Manhattan distance in the robot domain). The outcome returned by a run chain of algorithms consists of the fraction of successfully solved examples. Furthermore, the summed timeout of every search procedure is looked at.

The formula of the first proposed fitness function developed (the strong one) can be seen in equation 1, where w_s is the success weight and $w_s \in [0; 1]$, w_t is the time weight and $w_t \in [0; 1]$, $succ$ is the fraction of successfully solved examples, and $time$ is the total timeout as defined above. In the experiments, both of the weights are set to 0.5. To avoid rewarding synthesizers that quickly arrive at incorrect solutions, the score for the timeout (awarding search procedures that are fast) is only awarded if the evolved program manages to successfully solve all the given examples.

$$fit_{strong} = \begin{cases} w_s * succ & \text{if succ} < 1 \\ w_s * succ + w_t * \frac{1}{time} & \text{if succ} = 1 \end{cases} \quad (1)$$

It is important to notice that the *time* is the upper limit of how long the execution can take per task and the search procedures cease execution the moment a suitable program is found. In the earlier versions of the genetic algorithm, the actual execution time was considered as a metric for evaluation, which resulted in a discrepancy between what the chromosome represented (genotype) and how it was reflected in the behavior of a search synthesizer (phenotype). Due to this ambiguity, sequences with longer execution times were oftentimes favored over shorter ones, which obstructed the genetic process.

For the reasons, that should become more evident in the next section of this paper, one more fitness evaluation method has been considered (the weak one), as seen in equation 2, and applied to the tasks from the String and Pixel domains. Contrary to the first one, this method focuses on finding a balance between the number of solved tasks and speed instead of aiming to solve all the tasks. Both of the proposed fitness functions will be evaluated in the next section.

$$fit_{weak} = \begin{cases} \frac{succ}{time} & \text{if succ} \geq 0.5 \\ 0 & \text{if succ} < 0.5 \end{cases} \quad (2)$$

With fitness calculated, another crucial stage of the genetic algorithm is employing a selection strategy. As described in the related work section, Tournament Selection seems to be the most applicable, given the relatively small size of the population. Because of this and the promising results in an initial evaluation, this method has been chosen as the selection strategy.

3.4 Genetic operators

Two simple methods of recombining the parents have been examined - one-point crossovers[10] and two-point crossover [13]. In the one-point variant, two chromosomes are cut in one random point each and their genotypes are exchanged. Unfortunately, since the resulting exchanged parts can be of different sizes, this may lead to the creation of extremely long chromosomes. To bypass this obstruction, two-point crossovers are implemented, which are proven to give an improvement compared to one-point crossovers [13]. Now, both chromosomes are cut at two points with equal distance between them, and the cut-out fragments are swapped. This way, recombination is performed without changing the initial size of the chromosomes.

In the case of mutation operators, a vaster range of methods have been employed. The implemented mutations are:

- re-sampling a search procedure of one random gene
- re-sampling the execution time of one random gene

- adding a new random gene to the chromosome
- removing one of the genes from the chromosome
- multiplying the execution time of one random gene by a constant

To ensure that radical mutations are less likely to take place, the first four mutations have 12.5% chance of occurring if a mutation takes place. The last one, being the least revolutionary, has 50% chance of happening.

3.5 Final evaluation of selection procedures

To assess how well different genetic methods perform, I measure how fast the genetic algorithm converged to the final results and what the capacity of those results is. In other words, I verify how efficient different meta-parameters (such as the probability of mutations and fitness functions) are based on the time the entire evolution process took and how well the best-obtained program performs. That way, I can select the best-performing parameters and operators.

4 Experiments and Results

To assess how well different search algorithms perform and provide a valid comparison to my experiments, I first measure the execution time of separate search algorithms in each of the domains. Then, several experiments are performed with the main focus put on selecting appropriate values for mutation probabilities and comparing two different fitness functions, since both of those factors are very difficult to deduct based solely on the literature study. With these means, I can verify if the solution proposed in the methodology section is functional and answer the research question of whether it is possible to evolve a search procedure for the program synthesizer.

4.1 Robot path-planning

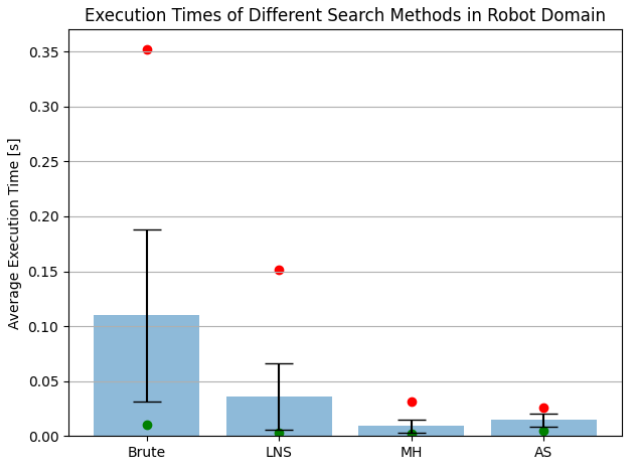

Figure 2: Average execution times of single algorithms in the robot path-planning domain, which managed to successfully solve a task. Green dots denote the minimum time, in which a search algorithm managed to solve a task. The red dots denote the maximum time. LNS stands for the Large Neighborhood Search, MH for Metropolis-Hastings, AS for A*.

Robot path-planning domain is the simplest and the least computationally demanding of all the tested domains. Therefore, it provides a good test ground for parameter tuning. The main goal is to steer the robot around a square board so that it picks up the ball and delivers it to the goal location. To calculate how close we are to the solution, an improved loss heuristic is used as described in [5]. Execution times, understood as the average amount of time a search algorithm needs to find a successful program, can be seen in figure 2. I decided to exclude the VanillaGP algorithm from the evaluation in this domain as its efficiency was incomparably worse than the rest of the algorithms. Each of the selected algorithms managed to solve each task. However, the execution times differ greatly with A* and Metropolis-Hastings exceeding Brute and Large Neighborhood Search in terms of performance.

The genetic procedure has been first run with the settings described in the methodology section. The tournament selection procedure has been used with a crossover rate of 0.8. Based on the execution times from the figure 2, the initial execution times of random search procedures have been sampled from a uniform distribution ranging from 0 to 0.1 for each of the search procedures. The size of generations was set to 100, to provide sufficient variety in the genotypes, along with the total number of generations of 50 and the maximum length of a chromosome sequence of 6. The evolution has been carried out on a set of 50 tasks of varying difficulty.

Mutation probability is one of the parameters, which value could not be directly derived from previous papers because of its dependence on the kind of genetic operators used and the domain. My initial hypothesis stated, that, since the kinds of applied mutations are not very radical (i.e. the degree of gene modification is not high), the mutation probability should be higher than the one proposed in the papers (from 0.01 to 0.05).

To verify this statement, for different values of this parameter ranging from 0.01 to 0.2, several evolution processes have been run. In order to compensate for the randomness of the evolution, those experiments have been repeated 10 times and the results for each value of the mutation probability have been averaged. The best-performing programs found for each of the parameters can be seen in the table 1. Each of them developed a search procedure that managed to find programs that solve all tasks within both the training and the test set.

p_m	Sequence	Timeout (ms)
0.01	A*	26.3
0.05	A*	26.4
0.1	Metropolis-Hastings	20.3
0.2	Metropolis-Hastings	19.5

Table 1: Sequences of algorithms evolved for different values of mutation probabilities (p_m) in the robot path-planning domain. Timeout is defined as the time limit for each of the algorithms.

Unfortunately, no interesting chains of search procedures have been discovered as the best-found programs simply consist of two single search procedures which already scored the best performance in the evaluation seen in figure 2. Those results do not come as surprising, as the tasks from the robot do-

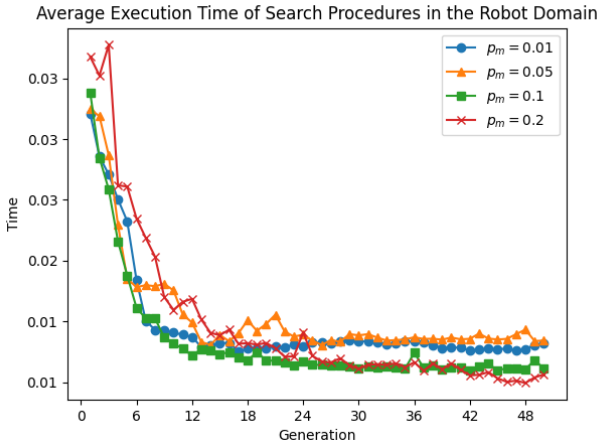

Figure 3: Average execution times of program synthesizers which found successful programs in the robot path-planning domain over generations. Different values of mutation probability parameters are tested.

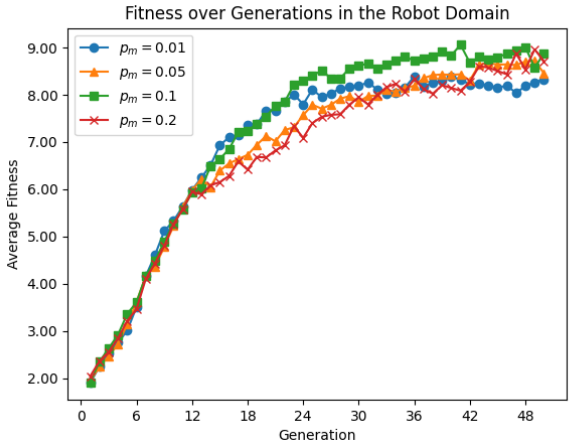

Figure 4: Average fitness of individuals in the robot path-planning domain over generations. Different values of mutation probability parameter are tested.

main are relatively uncomplicated. A combination of several search procedures can be expected to introduce an unnecessary level of complexity and overhead (each algorithm needs to be set up beforehand) in the areas, where near-optimal solutions can be found using more elementary methods.

What is interesting, however, is that the results fully support the proposed hypothesis, as they indicate that higher mutation ratios lead to finding more efficient search procedures. To analyze the course of the evolution, fitness and execution time over generations is depicted in figures 3 and 4. As seen in the graphs, synthesizers with lower mutation rates struggle to explore the whole search space and quickly settle for a less efficient solution, which is A* instead of Metropolis-Hastings. On the other hand, higher mutation rates are reflected in more chaotic and random behavior, which prevents the search from being stagnated in local minima. While the results do not differ widely, it looks like the mutation probability of 0.1 manages to find a perfect degree of chaos without descending into a fully random and chaotic search. This claim has been confirmed by the experiments performed in two other domains.

4.2 String transformations

String transformation aims to find a set of operations that need to be performed on a string of characters to transfer from one state to another. Finding an efficient search algorithm for the tasks derived from the string transformations domain is not a trivial job, which is pictured in figure 5. The graph demonstrates, that finding a solution in this domain is much more challenging than in the case of the robot path-planning domain. Moreover, A* greatly eclipses all the other considered domains in terms of performance, as it is the only search procedure that managed to solve all the training tasks within 10 seconds. What is more, not a single search procedure managed to achieve 100% accuracy on the test set.

Due to the aforementioned difficulties, the other form of fitness evaluation is tested- a weak one, as opposed to the strong form of fitness evaluation proposed before. Instead of focusing on finding programs that manage to solve all tasks and then optimize the execution time, I focus on solving as many tasks as possible in the shortest period. To that end, a new fitness formula is developed as seen in equation 2. Now, the genetic algorithm aims to find synthesizers that develop programs that are fast and manage to solve a satisfactory amount of tasks. The minimum success rate of the program is 0.5, however, this constant can be set to different values depending on what fraction of successfully solved task is deemed sufficient. The idea behind this approach makes sense, as in most cases what is looked for in program synthesis are solutions that are proficient at discovering the user intent and, at the same time, do not take too long to find satisfactory programs.

With that in mind, a new testing environment has been set up. Due to the expansive computation costs, the limit of generations has been set to 30 with 30 individuals in each generation and a maximum sequence size of 6. Based on the execution times of individual search methods, the initial generation has been populated randomly with execution times uniformly distributed in the range from 0 to 10. Tournament Selection Procedure has been used with the fitness defined in the equation 2. Crossover probability has been set to 0.8. To calculate loss, optimized string alignment has been utilized. The evolution has been carried out on a set of around 150 tasks with multiple examples each and the accuracy was evaluated on a separate set of 15000 tasks. Once again, a variety of mutation probabilities is tested.

The most efficient test procedures can be seen in the table 2. What truly stands out in those results is the fact that, with the mutation probability of 0.1, I managed to find a combination of two search procedures that performed better than their counterparts composed of a single search procedure. As it turns out, using Brute as the first search procedure followed by A* results in a search procedure that manages to solve

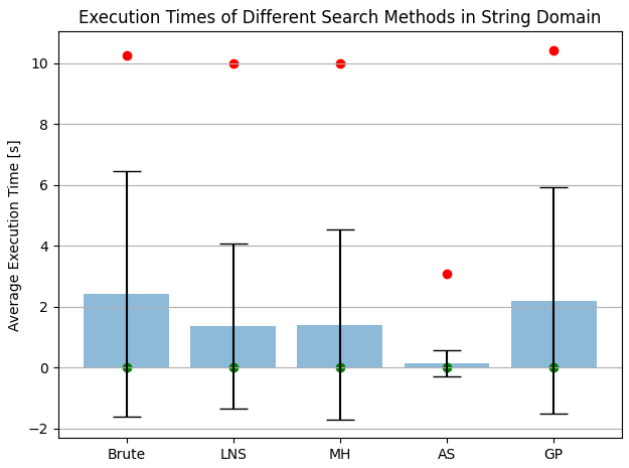

Figure 5: Average execution times of single algorithms in the string transformations domain. Green dots denote the minimum time, in which a search algorithm managed to solve a task. The red dots denote the maximum time. The maximum execution time per task was set to the maximum of approximately 10 seconds.

p_m	Seq (ms)	Time (ms)	Acc (%)
0.01	Brute 51.5 → Brute 51.5	103	50.6
0.05	Brute 40.7 → Brute 47.1	87.8	52
0.1	Brute 38.96 → AS 56.74	95.7	64.5
0.2	Brute 52.9 → AS 31.6	84.5	53.8

Table 2: Sequences of algorithms evolved for different values of mutation probabilities (p_m) along with timeouts for each procedure in a chain. Time column represents the summed timeout. Acc stands for accuracy which described the average of percentage of tasks solved by the best programs.

around 65% of train tasks with an average time of 95.7 ms per task, which results in the highest fitness among all four evolved chains. One very likely explanation can be found if we look at the greedy nature of Brute which is heavily utilized in the early stages of the search. First, the search space is trimmed to then focus on a smaller part of it with a more exploratory A*.

To validate the obtained results, the synthesized search procedure mentioned above has been run on a more sizeable test set, consisting of similar tasks with completely different examples. Two other search chains made of just one single search procedure have also been evaluated to provide a solid comparison. The results can be seen in table 3. It can be noticed that, even though a single A* procedure took shorter on average to complete tasks, the combination of Brute and A* managed to solve a larger amount of tasks. The performance of this combined search procedure stands out even more if one takes a look at a single Brute search, which falls short in both the execution time and the average solved number of tasks. This example clearly proves that one can evolve combinations of search procedures, that perform better than their singular equivalents.

Seq (ms)	Avg Time (ms)	Acc (%)
AS 38.96 → AS 56.74	14.92	52.48
Brute 38.96 → Brute 56.74	30.08	51.38
Brute 38.96 → AS 56.74	22.25	65.97

Table 3: Comparison of average execution times and accuracy of three different search procedures performed on a test set. One consisting only of Brute, one only of A*, and one as a combination of both.

4.3 ASCII (Pixel) art

This domain is especially interesting as both approaches to fitness calculation can be tested. The main goal here is to color a board consisting of black and white pixels in a certain way, that follows a specific pattern. As a loss function, an optimized combination of Hamming and cursor (which is used to recolor the pixels) distance is used. Some of the search procedures (Metropolis-Hastings and A*) manage to find fully accurate programs with a time limit of 60 seconds per task. However, their execution times are hardly acceptable. As seen in the figure 6, the dominance of A* over the other search procedure is even more noticeable than in the case of the previous domain. Therefore, I did not suspect to find any other appealing combinations of search procedures. However, this test case can still make for a good comparison and a bridge between the two fitness methods that were proposed earlier.

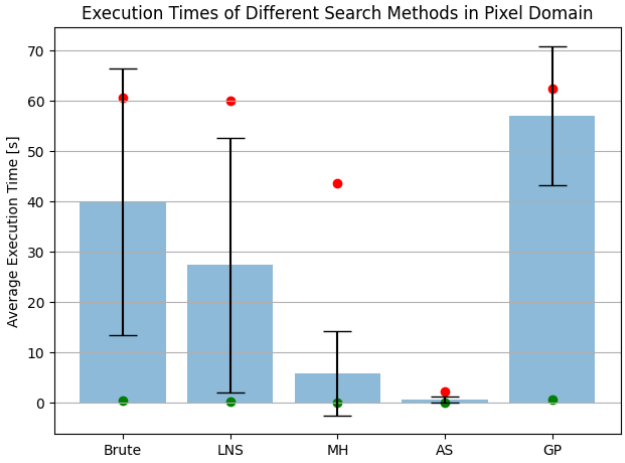

Figure 6: Average execution times of various search algorithms in the pixel art domain. The maximum task of execution per task was set to the maximum of approximately 60 seconds.

With that in mind, two experiments have been set up, with two different fitness formulas introduced before. To assure that the comparison is fair, all the other parameters are identical in both cases. The execution times of the initial generation have been sampled from a uniform distribution ranging from 0 to 10. Each generation is of size 30 with the limit of generations also set to 30 and the crossover probability of 0.8. The operators and other procedures have been configured as described in the methodology. The experiments have been run

several times with a mutation probability of 0.1, which was verified experimentally in the robot domain as the most suitable choice. Two different fitness evaluation methods mentioned before have been tested. Table 4 presents the results obtained on the train set, whereas table 5 depicts the results from the test set.

Fit	Search	Timeout (s)	Avg_t (s)	Acc (%)
Strong	AS	2.267	0.705	100
Weak	AS	0.501	0.361	50

Table 4: Comparison of results returned by the genetic algorithms running two different fitness functions on the train set in the pixel art domain. Fit represents the type of fitness equation used. The Avg_t column denotes the average time of execution per one task.

Fitness	Search	Timeout (s)	Avg_t (s)	Acc (%)
Strong	AS	2.267	0.721	99.1
Weak	AS	0.501	0.366	47.8

Table 5: Comparison of results returned by the genetic algorithms running two different fitness functions on the test set in the pixel art domain.

Although the weak fitness function results in a maximum execution time more than four times greater than its weak equivalent, the percentage of the solved task is only two times higher compared to the weak fitness function. This is most likely caused by the fact, that most tasks can be solved with a relatively small amount of effort, while a few outliers cause the discovery of a chain of search procedures that solves all the examples to be relatively long. What is more interesting, this property is also reflected in the average execution times, because, despite the high difference between the maximum execution times in the two examined cases, average execution times seem to be directly proportional to the number of solved tasks.

The results seem to confirm the statement made when introducing the second fitness evaluation method. Both fitness methods can be considered valid and their suitability relies on the preference of the user, who needs to answer the question of whether sacrificing half of the tasks is worth the difference in the execution time. Certainly, the fitness functions can be modified to find other optimal success/time ratios. Moreover, an interesting next step could be restructuring the strong fitness function to start awarding fitness for execution time only above a certain threshold of successfully completed tasks.

5 Conclusion and Future Work

To answer the main posed research question, I think that the experiments performed here, especially the one in the string domain, have proven that it indeed is possible to develop an efficient, combined search procedure with the use of genetic algorithms. However, the results are strongly dependent on the domain, considered search procedures, and what is exactly meant by an 'efficient search procedure'. To reflect the last dependency, two different fitness functions have been presented. While one of them maximizes the number of tasks solved at the cost of the synthesizer's speed, the other one aims to find an ideal balance between the satisfactory execution time and the percentage of the solved tasks. In the case of the former formula, all of the evolved selection procedures simply consist of one, the most efficient search procedure. Whereas the latter manages to prove the point of the research by providing a combination of two search procedures, namely Brute and A*, that attains higher results than in the instance of those two methods running separately in one of the domains.

On the other hand, the fact that multiple evolved procedures consist of just one search does not indicate that those results are to any degree less valuable. One could still employ the techniques described here with the fitness function tailored to their needs to find an acceptable balance between the time one search procedure is run and the number of tasks from a domain solved. Therefore, the range of applications is not restricted by the requirement of having more than one search procedure.

In the process of this research, I have proposed and thoroughly described a method to evolve a search procedure for program synthesis and answered all the subquestions related to the design specifications of the genetic algorithm. While most of the parameters and the operators have been chosen by carefully examining the existing literature, some of them, such as mutation probability, have been picked with the help of experiments in three different domains. All the details can be found in the methodology and experiments sections.

It is important to note, that there is still a lot of room left for future work, given the number of different factors that contribute to the final results. Starting from the type of initial timeout distributions (what if the distribution is normal instead of uniform?) to the types of selection strategies and crossover/mutation operators used, the range of possible augmentations is practically limitless and can be explored by other researchers. What is more, the created experimental framework is by no means bound by the search methods listed in this paper and can be easily extended with newer, more efficient ones. Analogically, new domains can be considered apart from the three of them discussed earlier, since other sets of tokens may be better suited for the sort of search procedures discussed here.

On a final note, one great limitation should be highlighted. A reader could notice that some of the search procedures were heavily under-performing compared to the others. This is mostly due to the fact, that they were still being worked on by another group of students as my research was performed, thus it is encouraged to repeat the experiments conducted here, once the search procedures have been improved.

6 Responsible research

During this research project, several guidelines have been followed to ensure that the research is conducted responsibly. Those guidelines include the topics that were discussed in the responsible research lecture. Since program synthesis is not directly associated with any ethical issues, the most important part of performing responsible research is ensuring that the final results are well-founded. The following section looks into all the steps I have taken to make the final results fully sub-

stantiated.

One of the greatest issues that haunts the scientific world is failing to replicate the results obtained by other researchers. In order to make my research fully reproducible, several measures have been adopted. Firstly, all the steps needed to conduct the same kind of experiments have been thoroughly listed along with the set parameters. Secondly, the full code, as well as the commit history and the used data sets, is publicly accessible through the Github platform [14]. Moreover, the specifications of the employed DelftBlue supercomputer [15] are detailed in the appendix .1.

Another important matter is ensuring that the used data set was valid and no data manipulation took place. The data used consisting of input-output examples from different domains is freely available on the Github repository. To assure that the data set is diverse enough, samples of many different sizes representing various difficulties of tasks were considered. Moreover, the data sets have been divided into train and test sets so that overfitting could be avoided.

Last but not least, both positive and negative results have been written down so that future researchers do not have to repeat the already made mistakes. Results from every domain have been scrupulously listed, no matter if they fell in line with the expected outcome or not.

References

[1] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. In *Foundations and Trends in Programming Languages*, volume 4, chapter Introduction. 2017. doi: 10.1561/2500000010.

[2] Armando Solar-Lezama. Introduction to Program Synthesis 6.S084/6.887 2020 - MIT course, 4 2022.

[3] Andrew Cropper and Sebastijan Dumancic. Learning large logic programs by going beyond entailment. In *IJCAI International Joint Conference on Artificial Intelligence*, volume 2021-January, 2020. doi: 10.24963/ijcai.2020/287.

[4] Stephen Muggleton and Luc de Raedt. Inductive Logic Programming: Theory and methods. *The Journal of Logic Programming*, 19-20(SUPPL. 1):629–679, 5 1994. ISSN 0743-1066. doi: 10.1016/0743-1066(94)90035-3.

[5] Bas Jenneboer and Sebastijan Dumanči. Program Synthesis with A*. Technical report, 2022.

[6] Stef Rasing and Sebastijan Dumančić. Improving Inductive Program Synthesis by using Very Large Neighborhood Search and Variable-Depth Neighborhood Search. Technical report, 2022.

[7] Victor Van Wieringen. Comparative analysis of the Metropolis-Hastings algorithm as applied to the domain of program synthesis. Technical report, 2022.

[8] Farhad Azimzade and Sebastijan Dumančić. VanillaGP: Genetic Algorithm for Inductive Program Synthesis. Technical report, 2022.

[9] Michael Negnevitsky. Evolutionary computation. In *Artificial Intelligence A Guide to Intelligent Systems Artificial Second Edition*, pages 219–257. 2005.

[10] David Beasley, David R. Bull, and R. R. Martin. An overview of genetic algorithms : Part 1, fundamentals. *University Computing*, 2(15), 1993. ISSN 0265-4385.

[11] Noraini Mohd Razali and John Geraghty. Genetic algorithm performance with different selection strategies in solving TSP. In *Proceedings of the World Congress on Engineering 2011, WCE 2011*, volume 2, 2011.

[12] Ahmad Hassanat, Khalid Almohammadi, Esra'a Alkafaween, Eman Abunawas, Awni Hammouri, and V. B.Surya Prasath. Choosing mutation and crossover ratios for genetic algorithms-a review with a new dynamic approach. *Information (Switzerland)*, 10(12), 2019. ISSN 20782489. doi: 10.3390/info10120390.

[13] David Beasley, David R Bull, and Ralph R Martin. An Overview of Genetic Algorithms : Part 2 , Research Topics 1 Introduction 2 Crossover techniques. *University Computing*, 15(4), 1993. ISSN 02654385.

[14] Michał Okoń, Fabian Radomski, Lucas Kroes, Nikolaos Efthymiou, and Philip Tempelman. Evolving Program Synthesizers. https://github.com/FabianRadomski/EvolvingProgramSynthesisers, 2022.

[15] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 1). https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1, 2022.

.1 System specifications

The experiments, including training and verification, have been run on the DelftBlue supercomputer, on the standard node with 2x Intel XEON E5-6248R 24C 3.0GHz CPU using 48 cores.

The scripts used to that end can be found on the GitHub repository and are named 'batchsingle.sbatch' (running single search procedures), 'searchevo.sbatch' (running the main evolution process), 'analyzeseraches.sbatch' (analyzing execution times of single search procedures).