Reward Based Program Synthesis for Minecraft

Adapting Program Synthesizers for Reward Evaluation and Leveraging Discovered Programs

Timur Mukminov

Supervisor(s): Sebastijan Dumančić, Tilman Hinnerichs

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Timur Mukminov
Final project course: CSE3000 Research Project
Thesis committee: Sebastijan Dumančić, Tilman Hinnerichs, Wendelin Böhmer

Abstract

Program synthesis is the task to construct a program that provably satisfies a given high-level specification. There are various ways in which a specification can be described. This research focuses on adapting the Probe synthesizer, traditionally reliant on input-output examples, to utilize reward-based synthesis. The generalization of Probe allows for flexibility in using various search algorithms, selection and updating algorithms, enhancing its applicability to a general case. By modifying the Probe algorithm to learn from rewards, we explore how exploiting existing programs as partial solutions impacts synthesis performance. Different ways of exploitation were tested, specifically, how much the probabilities change, and how a starting probabilities can affect the synthesis. Exploitation of programs could lead to faster synthesis but it could also lead to no solutions depending on the world environment.

1 Introduction

Program synthesis is an emerging field built on algorithmics, increasingly influenced by machine learning advancements. The concept of a computer program writing another program stands at the pinnacle of computer science with possibilities to advance automation and problem-solving, promising significant advancements in software development, scalability, and efficiency. Program synthesis has been an interest to computer scientists since the 1950s, during the initial surge of artificial intelligence research. Recent developments in computational power have reignited the passion for this field as it enables the synthesis of increasingly complex programs.

Reward-based synthesis is a nascent topic with relatively limited research compared to the more established example-based synthesis methods. While existing studies into Reward-Guided Synthesis (CUI et al, 2024)[1] have demonstrated promising results, it has yet to be addressed how generated or pre-existing programs could be utilised in the form of partial solutions or traces to affect the performance of reward-based synthesisers. Understanding this interaction is crucial for optimising synthesis algorithms to achieve faster program generation and optimal solutions. These priors could reduce the search space and induce bias towards the optimal program which will cause optimal solutions to be synthesised at a greater speed.

The supplementary objective of this study focuses on adapting the existing Probe synthesiser algorithm to use a reward-based evaluation system. This involves converting the original code into a different language, modifying the framework to integrate rewards into synthesis grammar. Followed by investigating how to adapt program synthesis to a game environment namely Minecraft.

The primary research aims to investigate how pre-existing successful programs and partial solutions found iteratively during the synthesis stage affect the performance of the Probe synthesiser. Specifically, the study seeks to optimise the synthesis time of the synthesiser by using pre-existing starting grammar found by evaluating a succesfful program and partial solutions to aid the discovery of a final solution.

By achieving these objectives, the study aims to provide insights into incorporating rewards into existing synthesis algorithms, and additionally provide insights into how a more exploitative algorithm affects the synthesis stage. Finally, this study will aim to contribute to the development of more advanced synthesis techniques and advancing the field of reward-based program synthesis.

This paper firstly discusses relevant background information in section 2, the generalisation and adaption of probe to Minecraft will be discussed in section 3, the research setup to exploit successful programs discovered and the results in section **??**. Furthermore, responsible research practices will be detailed in **??**, and the conclusion **??**.

2 Background

2.1 Program Synthesis

Program synthesis is the process of automatically generating computer programs from high-level specifications. This means a user describes generally what the program should do alongside a grammar that determines what the program can and cant do and then the program synthesiser creates the actual code to perform that task. Depending on the application, it could save time and effort, especially for complex tasks or repetitive coding such as the cell filling feature in excel called FlashFill (Gulwani, Sumit, 2011)[3]

Programming By Example

The most common approach to program synthesis is programming by example (PBE). For PBE the user provides specific input-output pairs to to guide the synthesis and determine the correctness of the generated program. The input-output pairs are used to search for a program which satisfies them and infers the desired program behaviour by matching or transforming the inputs into the outputs. PbE has various real-world applications. For example, PbE has proven to be a valuable paradigm for facilitating the automation of bioinformatics tasks (Gordon, P.M., Sensen, C. W., 2007; Wilkinson, M., 2006)[2][9]. Furthermore, PbE is evolving through combinations with machine learning strategies, such as the development of Neural Programming by Example (NPbE), which leverages deep neural networks for enhanced problem-solving capabilities (Shu, C., Zhang, H. 2017)[8].

This is the traditional approach as the input-output examples are easy to generate or are already available in large quantities which are used to evaluate the correctness of programs generated by the synthesiser. PBE is currently the leading approach to synthesising programs which has yielded many advancements in generating correct and efficient code.

Programming by Reward/Reward-based evaluation

An alternative method inspired by reinforcement learning is reward-based synthesis. This approach provides reward feedback based on the correctness of the program generated, making it more flexible and powerful, particularly in complex

domains where exhaustively generating programs to satisfy input-output example pairs within an appropriate time frame is unfeasible.

Recently, there has been growing interest in reward-based evaluation for program synthesis. Instead of relying on examples that need to be satisfied, the synthesiser receives a reward for creating a correct program, thus generating a feedback loop go improve its program generation. A key aspect of this technique is that it does not require detailed input-output pairs. Instead, an evaluation function assesses the correctness of the generated programs. This allows for easier application in scenarios where specifying examples is challenging, but defining a correct solution is straightforward. For example, in this study, a distance function determines the reward which cannot be transformed into an input-output example.

2.2 Synthesising Programs with Probe

One such advanced method is the Probe synthesiser. Probe employs techniques such as Just-in-Time (JIT) learning, a method that builds probabilistic models during synthesis using partial solutions. Additionally Probe employs guided bottom-up search algorithm which uses bottom-up enumeration with guidance from probabilistic models. These techniques enhance the synthesiser's guidance to bias towards the final program without requiring large amounts of prior examples

The algorithm can be described by two stages as seen in Figure 1:

- **Synthesis Stage:** The algorithm employs a bottom-up search strategy to build programs by combining smaller ones into larger ones. One of the cores of this algorithm is the Probabilistic Context-Free Grammar (PCFG) which helps guide the synthesis by enumerating the programs in the order of decreasing likelihood of appearing. This probabilistic guidance biases towards correct programs making them more likely to appear, resulting in a faster and more efficient synthesis process.
- **Learning Stage:** In this stage, the PCFG gets updated with new probabilities by learning from partial solutions found in the synthesis technique. This technique is exploited due to the observation that partial solutions often share syntactic similarities with the final solution. By rewarding the grammar nodes that were found in these partial solutions, the guidance provided by the PCFG gets biased to steer towards the final solution.

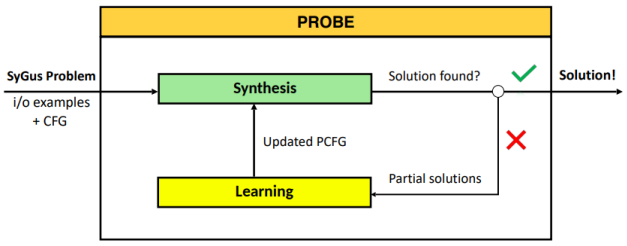

Figure 1: Overview of Probe algorithm (Shraddha Barke et al, 2020)[7]

Guided Bottom-Up Search

This is the synthesis stage where the algorithm employs a bottom-up search strategy to build programs by combining smaller ones into larger ones. The pseudocode can be found in the appendix B. One of the cores of this algorithm is the Probabilistic Context-Free Grammar (PCFG) which helps guide the the synthesis by enumerating the programs in the order of decreasing likelihood of appearing. This probabilistic guidance gives bias to programs more likely to appear resulting in faster and more efficient synthesis process.

Just-in-Time Learning

In this learning stage, the PCFG gets updated with new probabilites by learning from partial solutions found in the synthesis technique using the algorithm in figure 2. This technique is exploited due to the observation the partial solutions often share syntactic similarities with the final solution. By rewarding the grammar rules that were found in these partial solutions , the guidance provided by the PCFG gets biased to steer toward the final solution.

$$p(\mathrm{R}) = \frac{p_u(\mathrm{R})^{(1-\mathrm{Fit})}}{Z} \qquad \text{where} \qquad \mathrm{Fit} = \max_{\{P \in \mathrm{PSol} \mid \mathrm{R} \in \mathrm{tr}(P)\}} \frac{|\mathcal{E} \cap \mathrm{E}[P]|}{|\mathcal{E}|}$$

Figure 2: Formula for updating probabilities in PCFG (Shraddha Barke et al, 2020)[7]

3 Methodology

3.1 Generalise Probe

To enhance the flexibility and functionality of the Probe algorithm, the structure was generalised to be more modular. This allows for the use of different sub-algorithms and facilitates easy adjustment of various parameters. The pseudocode for the algorithm can be seen in (Shraddha Barke et al, 2020)[7]

Instead of being limited to guided bottom-up search alone, the modular approach allows interchangeability of different algorithms for searching the program space. It also facilitates straightforward redefinition or updating of selection and update functions.

The newly implemented program search algorithms have been implemented as iterators, enabling the return of intermediate results rather than only the final result. This property is particularly useful for reward-based synthesis, as it allows for continuous feedback throughout the synthesis process. Furthermore, this modification transfers the logic that monitors cycles from the guided bottom-up search algorithm to the Probe algorithm, therefore now allowing for adjustment of cycle lengths.

Probe was generalised to also use any iterator. However these iterators may lack evaluation caches and lists of partial solutions, therefore these were directly added to the probe algorithm. Consequently, every program returned by the iterator requires evaluation to fill the evaluation cache and partial solution store.

Algorithm 1 Generalised Probe algorithm

Input: PCFG G, set of input-output examples E, program iterator I, selection function SELECT, update function UPDATE, max. time T, cycle length C

Output: A solution P or $\perp$

Function PROBEGENERALISED(G, E, I, *SELECT*, *UPDATE*, T, C):
    $Eval\_Cache \leftarrow \emptyset$; // Initialise evaluation cache
    $State \leftarrow \emptyset$; // Initialise program iterator state
    while *time* $-$ *start_time* $< T$ **do**
        $P_{\text{Sol}} \leftarrow \emptyset$; // Initialise partial solutions
        while $i < C$ **do**
            $Program \leftarrow \text{ITERATE}(I, State)$; // Get next program
            $Result \leftarrow \text{EVAL}(P, E)$; // Evaluate program
            if $Result = E$ **then**
                return P ; // Solution found
            end
            if $Result \in Eval\_Cache$ **then**
                continue ; // Result already in evaluation cache
            end
            if $Result \cap E \neq \emptyset$ **then**
                $P_{\text{Sol}} \leftarrow P_{\text{Sol}} \cup Program$; // Add program to partial solutions
                $Eval\_Cache \leftarrow Eval\_Cache \cup R$; // Add result to evaluation cache
            end
        end
        $P_{\text{Sol}} \leftarrow \text{SELECT}(P_{\text{Sol}})$; // Select promising partial solutions
        if $P_{Sol} \neq \emptyset$ **then**
            $G \leftarrow \text{UPDATE}(G, P_{\text{Sol}}, Eval\_Cache)$; // Update grammar probabilities
            $Eval\_Cache \leftarrow \emptyset$; // Reset evaluation cache
            $State \leftarrow \emptyset$; // Reset program iterator
        end
    end
    return

3.2 Adapting Probe for Game Environments

As part of this research, the objective is to modify the existing Probe algorithm to incorporate rewards rather than traditional input-output pairs. This adaptation is necessary for implementing the algorithm in game environments such as MineRL (Guss et al, 2019)[4], where conventional input-output scenarios are absent. Instead, MineRL offers traces of real human actions to solve challenges, with rewards based on the progress made in these challenges.

The Minecraft[6] game, which MineRL is built on, is known for its expansive sandbox environment and procedurally generated worlds, each presenting unique landscapes and challenges for navigation. Amidst this diversity, environments can be controlled using seeds, allowing players to replicate identical worlds. This research utilises this mechanic to choose several seeds with unique landscapes such as flat plains, hilly desert, lush forests, and intricate cave systems, providing a varied spectrum of challenges for the synthesiser.

In this environment, the MineRL library provides rewards based on the task presented to the synthesiser and its evaluation of programs generated for navigating these challenges. A program is defined as a sequence of tuples of actions. An example would be walking forward 5 steps and walking left 3 steps. All required actions should be defined as grammar rules in the PCFG which the synthesiser uses to construct the sequence of actions to solve a task.

Language and Library Selection

The Julia language was chosen for its easy integration with Python, which is essential for running the MineRL environments. Additionally, the project utilizes and contributes to the leading library Herb-AI (Hinnerichs and Dumanci˘ c 2024)[5], developed specifically for program synthesis. As the original algorithm was written and optimized in Scala, which differs significantly in syntax and libraries from Julia, a complete rewrite of the algorithm was necessary. This included translating data structures, control flows, and models to their Julia/Herb-AI equivalents. Furthermore, a statistics library was employed to provide functions like "mean" without the need for manual implementation.

Reward-Based Evaluation

The algorithm was integrated with the API of the game to interact with the game environment, receive feedback, and dynamically calculate rewards. Without input-output examples, the algorithm shifted from evaluating the number of examples the partial solutions solve to a reward-based evaluation. The adapted algorithm now evaluates programs based on the rewards obtained in the game environment. This

required redefining the function for selecting partial solutions to choose a set number of programs which achieved the highest rewards. These then become the new partial solutions.

3.3 Exploiting Successful Programs

The second part of the research focuses on determining how to exploit successful programs which the algorithm has discovered. The following methods were used to explore the effect of changing the algorithm to exploit these programs:

- **Probability Updating:** Updating the grammar probabilities more aggressively based on successful programs to increase their likelihood of being selected.
- **Elitism:** Retaining the grammar created for the top-performing programs to bootstrap the synthesis process with previously successful programs.

4 Experiment Setup & Results

The primary goal of this research was adapting the existing algorithm, as was done for HerbSearch, and investigating the impact of exploiting discovered programs to aid the synthesis process.

4.1 Set-up

The code used for experiments is available on the *probe-with-minerl-exploit* branch of the HerbSearch GitHub repository [1]. Additional instructions on how to set up the system to run the code is available in the wikis of the same repository [2].

All experiments used an adapted *MineRLNavigateDense-v0* environment from version 0.4.4 of MineRL. The adapted version, *MineRLNavigateDenseProgSynth-v0*, adds the position of the agent to the observation space and adds the ability to send chat messages, which were used to send teleport commands to the start location, as well as disable mobs, food and health. Additionally, the randomisation of the compass which was used to calculate the rewards was turned off, implying the compass and rewards were always accurate. The adapted version can be found on the *prog-synth*[3] branch of the forked MineRL GitHub repository.

4.2 Hardware

The experiments were all run on the same system. The device used ran Ubuntu 24.04 LTS, had 8.0 GiB memory, 4-core i7-5600U and integrated graphics.

4.3 Environments - Seeds

For the experiments to be reproducible, 5 seeds were chosen that would generate the same world environment for each iteration of the experiment. This ensures that each iteration using the same seed generates the same world. In these environments, the goal is for the agent to reach a diamond block, which is 64 blocks away using Manhattan distance. The agent is given a reward based on how close it gets to the goal, with an additional 100 points awarded for reaching the block. Each environment has a unique maximum reward in the range of 60 to 75 excluding the extra points for reaching the destination. This reward mechanism is managed automatically by the Gym and MineRL libraries and cannot be changed to be normalised between seeds. The seeds were chosen to include diverse challenges for the algorithm to try to solve. Each seed has unique features which distinguish it from the other worlds. Each seed, their respective maximum reward, and title within the rest of the paper can be found in table 1.

World	Seed	Maximum Reward
World 1	958129	75
World 2	999999	64
World 3	6354	74
World 4	11248956	70
World 5	95812	68

Table 1: World Information

- 958129: Grassy terrain that is relatively flat with sparse trees to block the way. The agent spawns in front of a small cave opening with the destination being to the front left. Going forward will make you jump in it. It is possible to exit the cave opening by walking diagonally forward and left and jumping. Alternatively you can skip the cave opening by first walking left and then proceeding forward.
- 999999: Desert terrain that is relatively flat with no obstacles. Agent spawns in a desert and the destination is to the front and right.
- 6354: Forest terrain with many trees to block your way. The agent spawns right in front of a tree with the destination being to the front and left. It is possible to get on top of the trees or get into an alcove of trees where the only option is to reverse and go around.
- 11248956: Hilly terrain with a cave opening infront of agent. The goal is forward and to the left. Reward increases when entering cave but destination not inside the cave.
- 95812: Small hills and trees with a big hole between start and goal. Impossible to excape hold once entered.

These seeds will continue to be referred to as world 1-5 respectively.

4.4 Experiments

Each experiment was run 5 times and had a 15 minute timeout. The low number of reruns was due to each experiment taking a long time since each action has to be simulated in the environment. Each experiment used the same grammar, where the probabilities are displayed on the left and the actions on the right of the semicolon:

For the experiments, the fitness algorithm as described in Section 2 was changed. The higher the fitness for a grammar rule, the higher the chance of it being enumerated in the following synthesis cycle. Figure 3 shows the overall equations used for the fitness algorithms where x axis shows rewards, and y axis shows fitness.

[1] https://github.com/Herb-AI/HerbSearch.jl/tree/probe-with-minerl-exploit

[2] https://github.com/Herb-AI/HerbSearch.jl/wiki

[3] https://github.com/eErr0Re/minerl/tree/prog-synth

Listing 1: Minerl Grammar

```
minerl_grammar:
    1        :SEQ= [ACT]
    1        :ACT = (TIMES, Dict("move"
        => DIR, "sprint" => 1, "jump" =>
        1))
    0.125    :DIR = forward | back | left
         | right | forward-left | forward
        -right | back-left | back-right
    0.1666   :TIMES = 1 | 5 | 10 | 25 |
        50 | 75
```

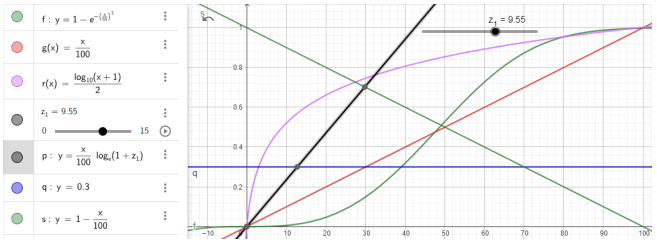

Figure 3: Fitness for reward of different algorithms]

Experiment 1

For experiment 1, a fitness of 0 was chosen meaning the probability of the grammar rules doesn't change. Because the probabilities doesn't change, the grammar rules get enumerated in order, meaning it first enumerates walking forward then backwards then left etc. which could lead to a correct direction to be enumerated last, every cycle. Unfortunately this experiment did not produce useful results for a baseline as every world wasn't able to find a solution in an appropriate amount of time and timed out.

Experiment 2

For this experiment, a constant fitness of 0.3 was chosen which meant the probabilities slowly got higher if the grammar rule was found in the partial solution. The same experiment was rerun with a fitness of 0.1, 0.5, 0.7, 0.9 which all oddly resulted in near identical results (ref appendix) therefore only fitness = 0.3 was chosen. The figures 4 and 5 shows us that we now have solutions for world 1 and 2. For world 1 the time to obtain 95% of the maximum reward by reaching within several steps of the goal was almost equal to generating a program that would travel those last steps. In world 3 it gets stuck in an alcove of trees several time where it requires to walk back and left to get out and increase the reward. This is represented by not having bar entry as it wasnt able to reach 95% reward. In world 4 the agent goes into a large hole because it gives the most reward however that hole is too deep to get out from. In world 5, the agent walks into the cave however it cannot get out as it would require the agent to walk backwards which decreases the reward. Since all subsequent experiments produced no result for world 4 and 5, the graphs for these worlds were left out.

Experiment 3

For the experiment 3, a linear fitness that was proportional to the best reward was chosen. For the rest of the paper best reward will be referred to as β. If the the grammar rule appeared in the partial solutions, the grammar was updated with a fitness of $\frac{\beta}{100}$. This fitness choice means that the earlier partial solutions that get a small reward don't affect the grammar as much as the larger almost correct partial solutions that receive large rewards. Figure 4 shows us that the time to generate a solution greatly decreased. This is partly due to the different probabilities causing the agent to take a more optimal route to the destination. This can be seen in world 1 where the agent skipped the 95% reward threshold and directly got to the reward. Figure 6 shows that World 3 now has a solution for obtaining 95% of the reward in under 700 seconds however it timed out before it could reach the goal.

The experiment was rerun but with mirrored fitness of $1 - \frac{-\beta}{100}$ so the closer you get to the goal, the less the grammar changes. The results timed out for every world likely because as you get closer to the reward you would require to change the grammar more aggressively to point you towards the goal. Although this was done only for experiment 3 so no conclusions could be made, it still serves as an indication of a correct approach.

Experiment 4

For experiment 4, a

$$fitness = 1 - exp^{(\frac{-\beta}{55})^3} \tag{1}$$

was chosen. the fitness is smaller at the lower values of reward and bigger at higher values compared to fitness algorithm in Experiment 3. Figure 3 shows us the fitness level compared to reward of the algorithms. Compared to experiment 3 it exploits less early on and exploits more closer to the goal. You can see in figure 4 and 5 that the time to generate a 95% correct solution decreased however the time take to generate a full correct solution increased and in the case of world 3 it wasn't able to generate a solution in an appropriate amount of time. This could mean that the majority of time for the program generation comes in the mid to late stages of the final solution so optimising for it will likely yield better results however it requires exploration to find the sequence of actions to reach the goal.

Experiment 5

For experiment 5, a Logarithmic fitness function has been defined as follows

$$fitness = log_{10}(\frac{1+\beta}{2}) \tag{2}$$

Such a fitness rewards greatly throughout the rewards range compared to the Linear fitness. Figure 4 tells us that it wasn't able to generate a solution for world 1 compared to the other methods. Worlds 2 and 3 saw a drastic increase in time to reach the 95% reward threshold

4.5 Experiment 6

For experiment 6, a more complex algorithm that uses 2 variables was chosen to decide the fitness of each grammar

rule. In particular

$$fitness = \frac{\beta}{100} * log(1 + num\_appearances) \quad (3)$$

was used, where the linear algorithm from experiment 3 was used except there is an additional variable that multiplies the fitness by the number of occurrences of the grammar rule in the entire program. This means higher occurrences of a grammar rule gets a higher fitness value. As seen from Figure 4 and 5 and 6 this fitness algorithm makes the synthesiser reach the almost correct program in around half the time, however because the grammar probabilities are highly tailored to reach as close as possible, getting to the block requires exploration which the probabilities highly discourage. This results in the complex algorithm bar in figure 5 where it reaches within a couple blocks of its destination significantly more quickly than other methods but the final couple blocks could take more than the same amount.

4.6 Results

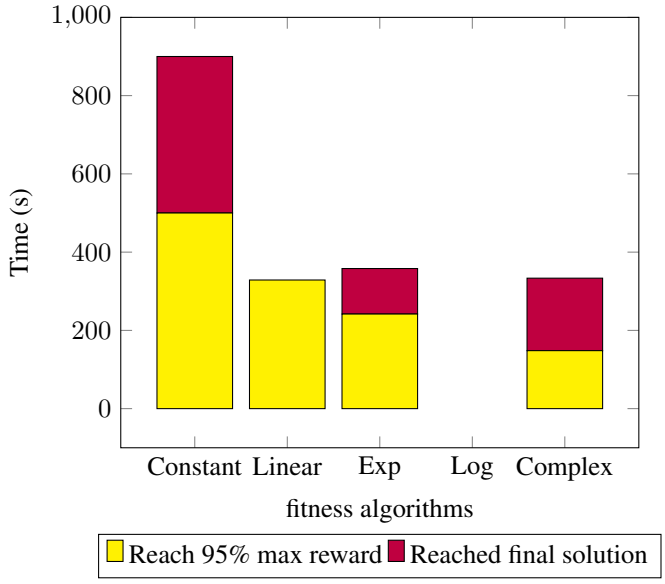

Figure 4: World 1. Time to find solution using different fitness algorithms. Missing Bars means no solution

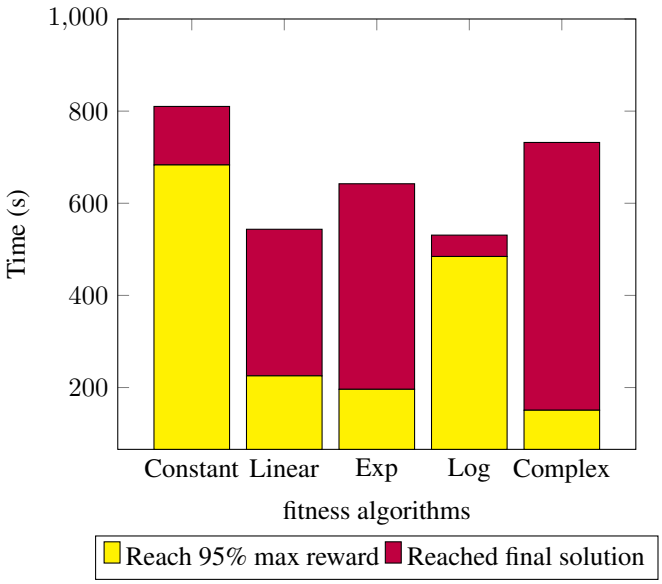

Figure 5: World 2. Time to find solution using different fitness algorithms

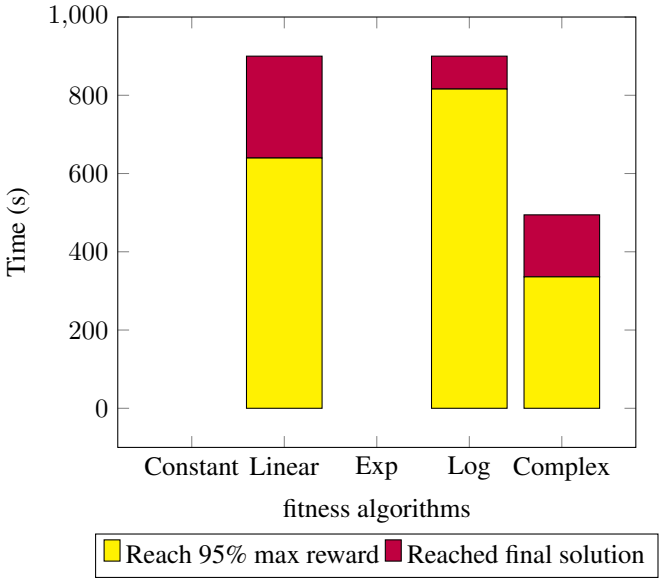

Figure 6: World 3. Time to find solution using different fitness algorithms. Missing Bars means no solution

4.7 Experiment 7

Experiments 2-6 were rerun, however, for each experiment another run was made beforehand to determine the probabilities of the grammar rule. These were then used in the starting grammar instead of a uniform probability model. This change could have several effects: since the starting grammar is now different, the synthesiser could find a different route which could be more optimal. Figure 7 shows it takes around 2.5 times less time to reach a solution in world 2 using the linear fitness algorithm compared to

starting with uniform probabilities in figure 5. With different starting conditions it is also now able to solve an additional seed namely world 4 as seen in figure 8. On the other hand it could also lead to unsolvable situations for example figure 10 shows that in world 3, when changing the starting conditions only the log fitness model was able to find a solution albeit 2 times faster.

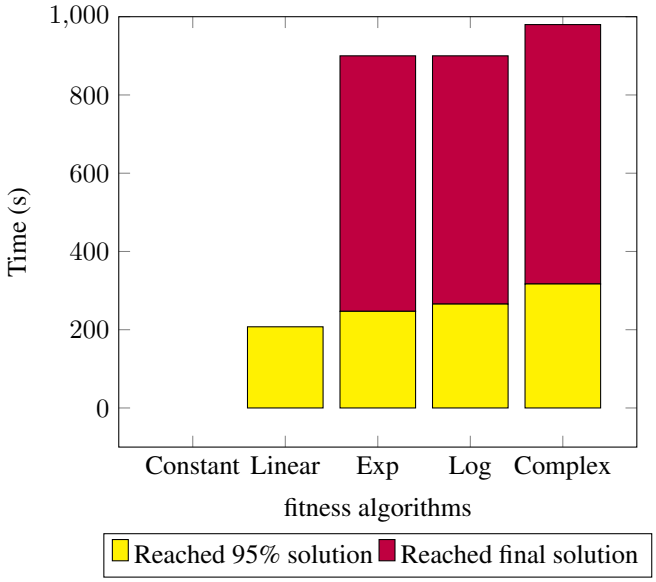

Figure 7: World 2. Time to find solution using different fitness algorithms and with starting probabilities

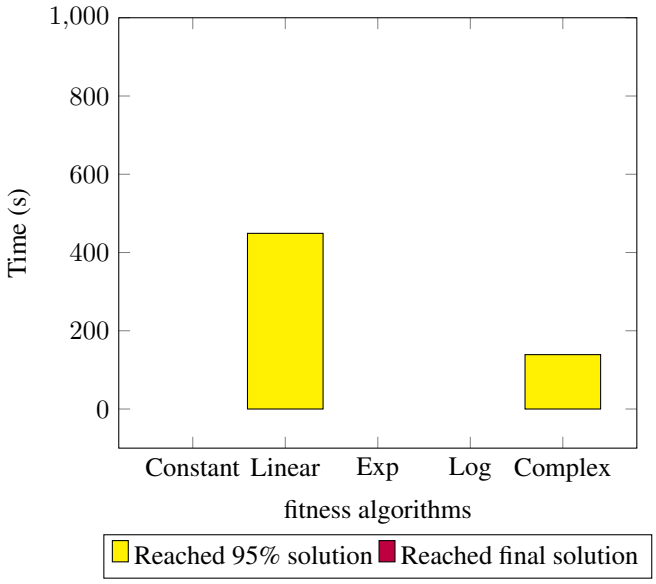

Figure 8: World 4. Time to find solution using different fitness algorithms and with starting probabilities

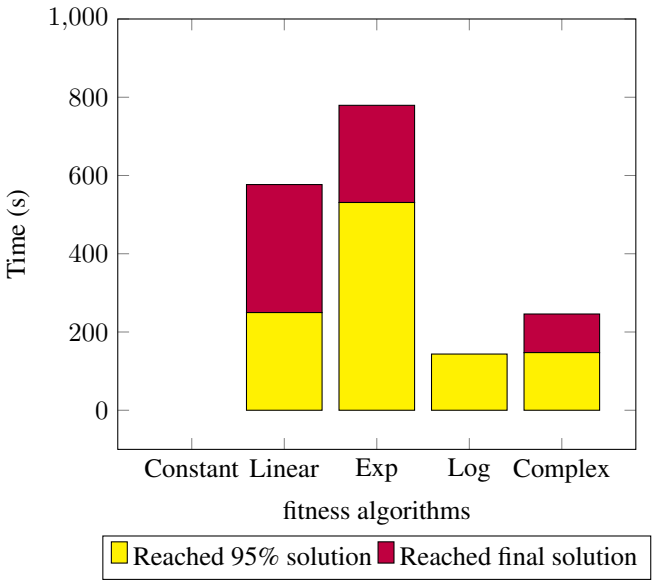

Figure 9: World 1. Time to find solution using different fitness algorithms and with starting probabilities

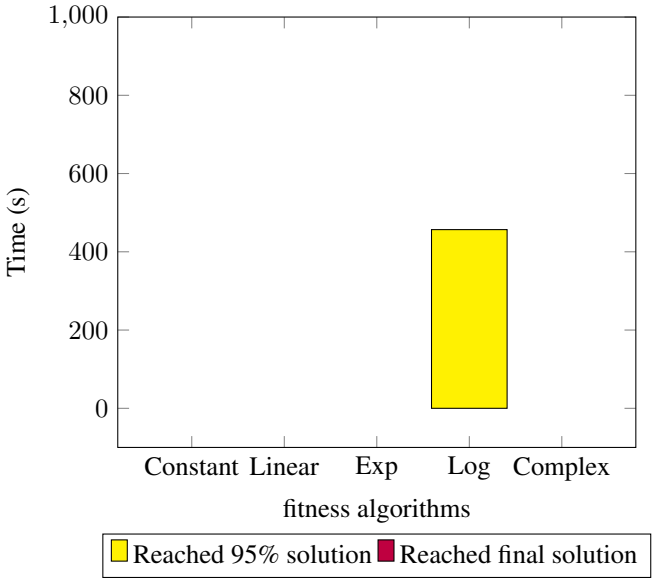

Figure 10: World 3. Time to find solution using different fitness algorithms and with starting probabilities

4.8 Trend in results

From experiments 2-6, there seems to be a trend where more exploitation of grammar rules that were previously used reduces the time to generate a solution whose feedback reaches 95% of maximum reward. However, there seems so be no trend in the in time to generate a full solution.

From experiment 7 , it is hard to conclude a trend. Depending on the world seed and fitness algorithm used, the synthesis time could either drastically reduce or increase. It is able to solve new worlds, but it is also not able to solve

previously solvable configurations. Although there seems to not be a trend on the general scale, simpler worlds like 1 and 2 seem to benefit as their time to generate a solution with 95% of the reward decreases across fitness algorithms. This suggests that simpler environments benefit from exploitation more than complex environments which is in line with the hypothesis. However across the fitness algorithms, it seems that using more exploitation has a negative effect on the synthesis time. This suggests that there exists an optimal trade off between exploitation and exploration per world basis.

The rest of the results and graphs can be found in the appendix A

5 Responsible Research

While the methodology of the original data collection lacks detailed documentation, our methodology over the course of our research will be thoroughly described to ensure reproducibility of our results using the same dataset.

The algorithm used for the generation of the Minecraft probe is deterministic meaning that it is guaranteed that running the same algorithm in the same Minecraft environment will consistently produce the same program in a similar amount of time accounting outside variables like temperature of device. This predictability of results allows for researchers to reproduce and verify our results. Additionally we meticulously document how to set up the environment and provide comprehensive instructions on how to run the algorithm to recreate the conditions that have produced our results.

6 Conclusions and Future Work

The goal of this research project was to further explore rewards as a method of specification in program synthesis. We generalised the already existing Probe synthesiser and then tailored it to synthesise for the environments in MineRL. We made changes to probe to allow the use of different selection methods. Changed Probe to utilises rewards from evaluation. This meant redefining partial solutions and how grammar gets updated.

We investigated the effect of changing the probabilities in a way that biased already discovered partial and full solutions to explore how exploitation affects the synthesis time. By changing the fitness algorithm we found that favouring the partial solutions with better rewards will decrease the time needed to reach the final solution, however it also has a negative effect of decreasing exploration which is needed to reach the final solution in an appropriate amount of time. Depending on the environment changing the fitness algorithm could change the grammars probabilities where a new route is taken which sometimes solves a new environment or sometimes reaches a point where its unsolvable.

There is a limited number of results available due to the duration of each experiment and the many world types available therefore a concrete conclusion on which exploitation methods is best cannot be made.

Building on the generalization and reward-based adaptation of the Probe algorithm, several paths to explore for future research:

- Balancing Exploration and Exploitation: finding the optimal threshold for amount of exploitation and exploration
- Explore hybrid approaches: explore how different approaches could be utilised based on feedback from environment (eg. if synthesis gets stuck in a local maxima)
- Trace evaluation: using traces of human gameplay to guide synthesis
- Exploring Different Domains: Extend the application to environments beyond navigation.

A Graphs

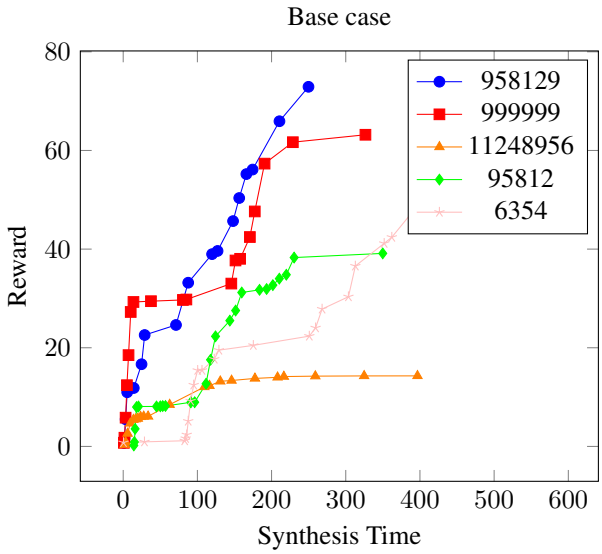

Figure 11: Base case, Partial solution: $reward > \beta$, Cycle length 6, Select 5 programs with highest reward, $fit = min(\frac{\beta}{100}, 1)$

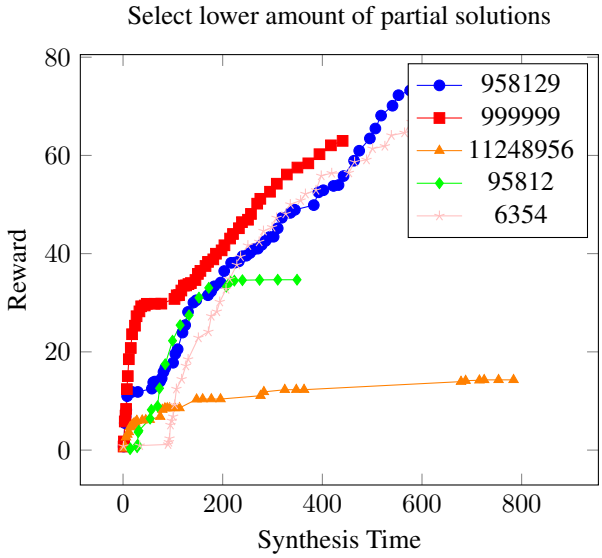

Figure 12: Select lower amount of partial solutions, Partial solution: $reward > \beta$, Cycle length 4. Select 3 programs with highest reward. Update based on last action. $fit = min(\frac{\beta}{100}, 1)$

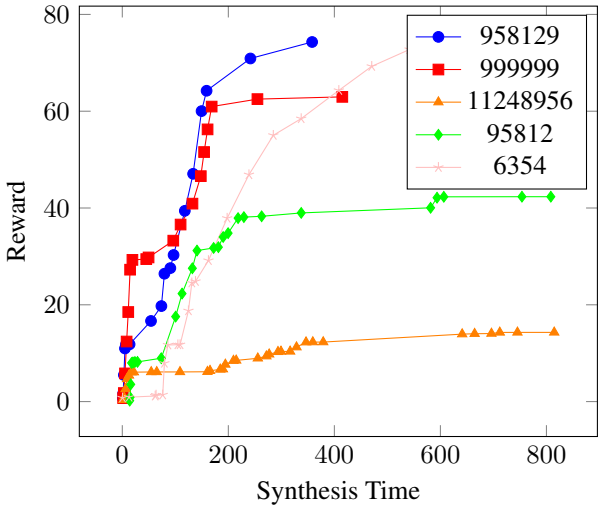

Figure 13: Use a different fitness algorithm, Partial solution: $reward > \beta$, Cycle length 6. Select 5 programs with highest reward. Update based on last action. $fit = 1 - exp^{(-(\frac{1}{mean\_reward})*\beta)}$

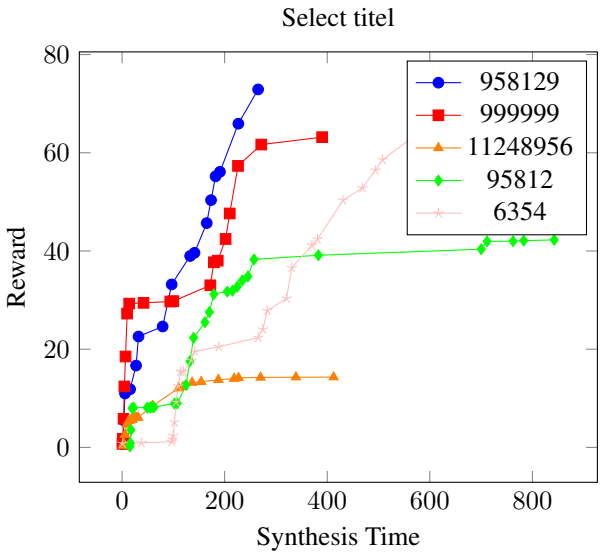

Figure 14: experiment 1 again: $fit = \frac{\beta}{100}$

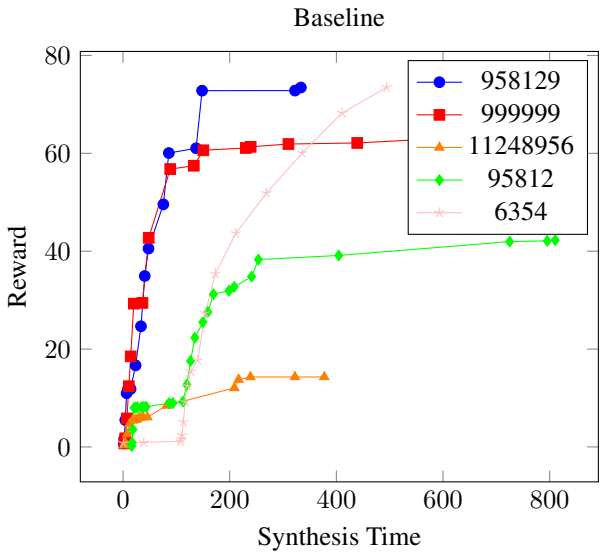

Figure 15: $fit = min(1, (\frac{\beta}{100}) * (log(1 + appearances)))$

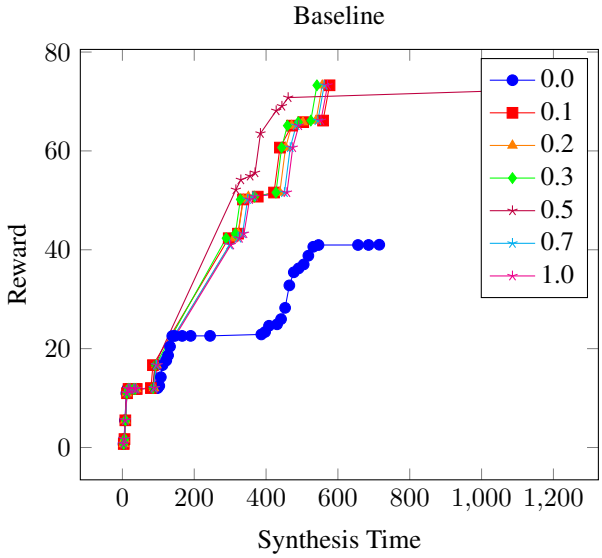

Figure 16: $fit = c$

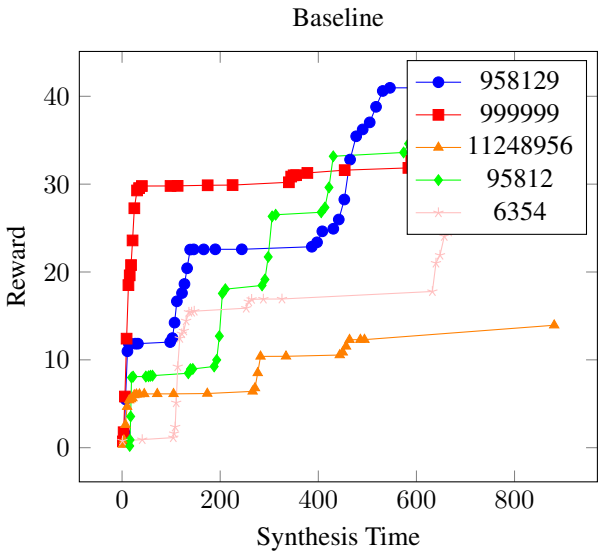

Figure 17: $fit = 0$

B Probe guided search algorithm

Algorithm 1 Guided Bottom-up search algorithm

Input: PCFG $\mathcal{G}_p$, input-output examples $\mathcal{E}$, and optionally, the initial state of the search
Output: A solution P or $\perp$, and the current state of the search

procedure GUIDED-SEARCH($\mathcal{G}_p, \mathcal{E}, \langle \text{LVL}_0, \text{B}_0, \text{E}_0, \text{PSol}_0 \rangle = \langle 0, \emptyset, \emptyset, \emptyset \rangle$)
  LVL, B, E, PSol $\leftarrow$ $\text{LVL}_0, \text{B}_0, \text{E}_0, \text{PSol}_0$ ▷ Initialize state of the search
  while LVL $\leq$ LVL_0 + LIM **do**
    for $P \in$ NEW-PROGRAMS($\mathcal{G}_p$, LVL, B) **do** ▷ For all programs of cost LVL
      EVAL $\leftarrow$ $[\langle i, [\![P]\!](i) \rangle \mid \langle i, o \rangle \in \mathcal{E}]$ ▷ Evaluate on inputs from $\mathcal{E}$
      if (EVAL $= \mathcal{E}$) **then**
        return $(P, \langle \text{LVL}, \text{B}, \text{E}, \text{PSol} \rangle)$ ▷ P fully satisfies $\mathcal{E}$, solution found!
      else if (EVAL $\in$ E) **then**
        continue ▷ P is observationally equivalent to another program in B
      else if (EVAL $\cap \mathcal{E} \neq \emptyset$) **then** ▷ P partially satisfies $\mathcal{E}$
        PSol $\leftarrow$ PSol $\cup P$
      B[LVL] $\leftarrow$ B[LVL] $\cup \{P\}$ ▷ Add to the bank, indexed by cost
      E $\leftarrow$ E $\cup$ EVAL ▷ Cache evaluation result
    LVL $\leftarrow$ LVL + 1
  return $(\perp, \langle \text{LVL}, \text{B}, \text{E}, \text{PSol} \rangle)$ ▷ Cost limit reached

Input: PCFG $\mathcal{G}_p$, cost level LVL, program bank B filled up to LVL $-$ 1
Output: Iterator over all programs of cost LVL ▷ For all production rules

procedure NEW-PROGRAMS($\mathcal{G}_p$, LVL, B)
  for $(\text{R} = \text{N} \rightarrow (t\ \text{N}_1\ \text{N}_2\ \ldots\ \text{N}_k) \in \mathcal{R})$ **do**
    if $\text{cost}(\text{R}) = \text{LVL} \wedge k = 0$ **then** ▷ t has arity zero
      yield t
    else if $\text{cost}(\text{R}) < \text{LVL} \wedge k > 0$ **then** ▷ t has non-zero arity
      for $(c_1, \ldots, c_k) \in \left\{ [1, \text{LVL}]^k \,\middle|\, \sum c_i = \text{LVL} - \text{cost}(\text{R}) \right\}$ **do** ▷ For all subexpression costs
        for $(P_1, \ldots, P_k) \in \{ \text{B}[c_1] \times \ldots \times \text{B}[c_k] \mid \bigwedge_i \text{N}_i \Rightarrow^* P_i \}$ **do** ▷ For all subexpressions
          yield $(t\ P_1\ \ldots\ P_k)$

Figure 18: Pseudocode of the Synthesis stage of Probe (Shraddha Barke et al, 2020)

References

[1] GUOFENG CUI, YUNING WANG, WENJIE QIU, and HE ZHU. Reward-guided synthesis of intelligent agents with control structures. 2024.

[2] Paul MK Gordon and Christoph W Sensen. Seahawk: moving beyond html in web-based bioinformatics analysis. *BMC bioinformatics*, 8:1–13, 2007.

[3] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 317–330, 2011.

[4] William H Guss, Brandon Houghton, Nicholas Topin, Philip Wang, Mathias Codel, Manuela Veloso, and Ruslan Salakhutdinov. Minerl: A large-scale dataset of minecraft demonstrations. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS) Competition and Demonstration Track*, 2019.

[5] T. Hinnerichs and S. Dumancić. Herb.jl: A library for defining and efficiently solving program synthesis tasks in julia. https://github.com/Herb-AI/Herb.jl, 2024. GitHub repository.

[6] Mojang. Minecraft, 2009.

[7] Hila Peleg Shraddha Barke and Nadia Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. page 29, November 2020.

[8] Chengxun Shu and Hongyu Zhang. Neural programming by example. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[9] Mark Wilkinson. Gbrowse moby: a web-based browser for biomoby services. *Source Code for Biology and Medicine*, 1:1–8, 2006.