Program Synthesis from Rewards with Probe

Adjusting Probe to Increase Exploration When Synthesising Programs from Rewards in Minecraft

Nils Marten Mikk[1]

Supervisors: Sebastijan Dumančić[1]**, Tilman Hinnerichs**[1]

[1]**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 22, 2024

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

Abstract

Program synthesis is the task of generating a program that satisfies some specification. An important aspect of program synthesis is the method of specification. There are various ways in which a desired program can be specified, such as I/O examples, traces, and natural language. This research paper aims to explore a novel method of specifying a desired program in program synthesis – rewards. This concept is explored by adjusting the Probe program synthesiser to solve the dense navigation environments in MineRL. In order to avoid local maxima, it is necessary to increase the amount of exploration. To that end, different ways of increasing exploration were tested by changing the parameters of Probe. By increasing the amount of exploration, it is possible to solve more environments, or solve them faster. But increasing exploration could also have the opposite effect, depending on the environment.

1 Introduction

Program synthesis is the task of generating a program according to some user-provided specification. It can be used in many situations, such as data transformation (Gulwani 2011; Singh and Gulwani 2016), code suggestions (Gvero et al. 2013), and code repair (Nguyen et al. 2013; Singh, Gulwani, and Solar-Lezama 2013). Therefore, program synthesis can be of great use to software engineers by helping them write code or fix bugs, increasing their efficiency.

The main idea of program synthesis is to search over the space of all programs until a program that satisfies the specification is found. Because the size of the search space is very large (possibly infinite), programs need to be generated in a smart way.

An important part of program synthesis is the method used for describing a desired program. The choice of specification can greatly influence the outcome of the program synthesiser. It can also affect the usability of the synthesiser. There are many ways in which the program can be specified, such as I/O examples, partial programs, formal specifications, traces, and natural language.

This project aims to fill a knowledge cap in the types of specifications used for program synthesis. It explores a novel method of specifying a desired program – learning from rewards. This could, for example, be used to generate programs for playing video games. In this paper, it is used to synthesise programs for playing Minecraft with the Probe program synthesiser.

Probe is a program synthesiser that uses a probabilistic context-free grammar to guide the search by enumerating programs with higher probabilities first. It can update the probabilities of the rules during execution using partial solutions, therefore requiring no training data. This synthesiser can be used to synthesise programs from rewards by updating the probabilities based on the rewards received.

The goal of the program synthesiser would be to generate programs that maximise the reward. While trying to maximise the reward, the synthesiser could get stuck in a local maximum and never reach the desired program. For this reason, it is important to find ways for the synthesiser to explore more novel programs that do not end up in a local maximum.

This paper attempts to answer the following questions:

- How to define program synthesis from rewards?
- How to adjust Probe to learn programs from rewards?
- How to increase the amount of exploration when learning from rewards with Probe?
- How does increasing the amount of exploration affect the time it takes to learn programs from rewards with Probe?

The following contributions are made in this paper:

- A generalised Probe synthesiser that enables the use of arbitrary search algorithms. It also allows for easily changing its parameters, such as the function for updating the grammar, or the number of programs to iterate in one cycle (section 3.1).
- Defining program synthesis from rewards (section 3.3).
- Adjusting the Probe synthesiser to learn programs for playing Minecraft from rewards by redefining partial solutions, observational equivalence, and the function for updating the grammar (section 3.4).
- Finding ways to increase the amount of exploration by changing the parameters of Probe and the grammar, and analysing their effect on the runtime. (section 4).

2 Background

This section gives an overview of the background information necessary for understanding this paper.

2.1 Program Synthesis

The goal of program synthesis is to automatically generate a program that satisfies a user-provided specification. This is done by searching over the program space – the set of all programs that can be derived from the provided grammar. The different search algorithms enumerate the programs in the program space until a program is found that satisfies the specification, or until a time limit is reached. The efficiency of program synthesis is mainly influenced by the choice of the search algorithm, method of specification, and program space.

Enumerative search orders the search space according to some metric, such as program size, and then enumerates the programs. Two common enumerative search methods are top-down and bottom-up tree search. Top-down tree search start enumerating from the starting symbol, while bottom-up starts from terminal symbols. Various methods for pruning can be used to arrive at the desired program quicker, such as observational equivalence (Albarghouthi, Gulwani, and Kincaid 2013). Enumerating programs can be improved by guiding the search with probabilistic models by exploring likelier programs first. The Probe synthesiser (Barke, Peleg, and Polikarpova 2020) makes use of this idea by using a probabilistic context-free grammar (PCFG) and enumerating programs with higher probabilities first. It can learn the probabilities during execution without requiring any training data, unlike previous approaches. Other approaches to program synthesis include component-based synthesis (Jha et al. 2010), genetic programming (Koza 1994), solver-aided programming (Torlak and Bodik 2013), and many more.

A common method of specifying the desired program is programming by example. In this specification, the desired program is given as a set of input-output examples. For every program that is enumerated, the program is evaluated using the input examples. A program is said to satisfy the specification if the outputs of evaluating the program on the input examples match the output examples.

2.2 Probe Program Synthesiser

The main idea of the Probe program synthesiser (Barke, Peleg, and Polikarpova 2020) is to learn the probabilities of the probabilistic context-free grammar during execution. This means that it does not require any training data for learning the probabilities.

Guided Bottom-Up Search For generating the programs, the Probe synthesiser makes use of the *guided bottom-up search* algorithm, introduced in the Probe paper. This algorithm takes as input a PCFG $\mathcal{G}_p$, a set of I/O examples $\mathcal{E}$, and the state, consisting of the starting cost level Lvl, the program bank B, the evaluation cache E, and the set of partial solutions $PSol$. The program bank contains previously generated programs organised by their cost, and the evaluation cache contains the results of previous evaluations.

The algorithm generates the programs level by level, using the rules from $\mathcal{G}_p$ and the programs from the bank. The programs generated in a level have a cost equal to the level. The cost of a program is defined as the sum of the cost of the rule and the costs of the subexpressions taken from the bank. The probability of a rule is converted to an integer as $\lfloor -\log_2(probability(rule)) \rceil$ so that it can be used as a cost. For each program, it evaluates the program with the inputs of $\mathcal{E}$ and checks if the results match the outputs of $\mathcal{E}$. If they do, it returns the program, along with the state. Otherwise, it checks if the program is observationally equivalent to another program in the bank by comparing the result with the results in E. If it is not equivalent, the program is added to the bank and the result to the evaluation cache. The program is also added to the list of partial solutions if it correctly solved some examples. This process repeats for one cycle – until a certain number of levels have been searched. If the correct program was not found, it returns nothing as the program, and the state.

Probe Algorithm The Probe algorithm takes as input a context-free grammar (CFG) $\mathcal{G}$ and a set of I/O examples $\mathcal{E}$. Firstly, it changes the CFG $\mathcal{G}$ to a PCFG $\mathcal{G}_p$ with uniform probabilities. It then initialises the state by setting the level Lvl to 0, and the bank B and evaluation cache E to the empty set. The guided bottom-up search is then run for one cycle. If the program is found, it returns the program. Otherwise, it selects promising partial solutions from the set of partial solutions $PSol$. In the Probe paper, they defined three different methods for selecting promising partial solutions: largest subset, first cheapest, and all cheapest. If the set of promising partial solution is not empty, the probabilities of $\mathcal{G}_p$ are updated, and the search state is reset. Otherwise, the search is continued from the same state with the same probabilities. This process continues until a time limit is reached. The pseudocode for the synthesiser can be seen in algorithm 1.

The update function gives higher probabilities to rules that are part of partial solutions that solve many examples. The probabilities of each rule R are updated according to the formula:

$$p(R) = \frac{p_u(R)^{1-FIT}}{Z}$$

where

$$FIT = \max_{\{P \in PSol | R \in tr(P)\}} \frac{|\mathcal{E} \cap E[P]|}{|\mathcal{E}|}$$

where p_u is the uniform distribution of $\mathcal{G}_p$, Z is the normalisation factor, $tr(P)$ is the set of rules used in deriving P, and $E[P]$ is the result of evaluating P.

2.3 Minecraft and MineRL

The sandbox video game Minecraft is used to explore synthesising programs from rewards. It is a game with three-dimensional procedurally generated worlds made of blocks. In this game, players can explore the world, gather resources, craft items with the resources, build structures, and much more.

Algorithm 1: Probe algorithm (Barke, Peleg, and Polikarpova 2020)

Input: CFG $\mathcal{G}$, set of input-output examples $\mathcal{E}$
Output: A solution P or $\perp$

function PROBE($\mathcal{G}, \mathcal{E}$)
  $\mathcal{G}_p \leftarrow \langle \mathcal{G}, p_u \rangle$ ▷ Initialise PCFG to uniform
  $Lvl, B, E \leftarrow 0, \emptyset, \emptyset$ ▷ Initialize search state
  while not timeout **do**
    $P, \langle Lvl, B, E, PSol \rangle \leftarrow$ GUIDEDSEARCH($\mathcal{G}_p, \mathcal{E}, \langle Lvl, B, E, \emptyset \rangle$) ▷ Search with current PCFG $\mathcal{G}_p$
    if $P \neq \perp$ **then**
      return P ▷ Solution found
    $PSol \leftarrow$ SELECT($PSol, E$) ▷ Select promising partial solutions
    if $PSol \neq \emptyset$ **then**
      $\mathcal{G}_p \leftarrow$ UPDATE($\mathcal{G}_p, PSol, E$) ▷ Update the PCFG $\mathcal{G}_p$
      $Lvl, B, E \leftarrow 0, \emptyset, \emptyset$ ▷ Restart the search
  return $\perp$

The MineRL Python library is used to interact with the game. It provides several environments with different tasks, and gives the possibility to define your own environments. For example, it provides an environment with the goal of obtaining 64 blocks of wood by chopping trees. For each wood the agent obtains, it gets a reward. Using this reward and the observations returned from the environment, such as the RGB image and contents of the inventory, it is possible to guide the agent towards the end goal. For each environment, a set of possible actions is defined as a dictionary of action-value pairs. For example, to move the agent forwards, we can set `action["forward"]=1` and call the step function. The step function will run one time step of the environment with the provided action.

Many approaches have been used to solve the MineRL environments. The approaches used among the finalists of previous MineRL competitions include Sample-efficient Hierarchical AI (Mao et al. 2022), Hierarchical Deep Q-Network (Skrynnik et al. 2021), imitation learning (Amiranashvili et al. 2020), and behavioural cloning (Kanervisto, Karttunen, and Hautamäki 2020). Almost all of them use reinforcement learning in some way, unlike the approach taken in this paper.

3 Methodology

The research project is split into four steps:

1. Generalising the Probe program synthesiser.
2. Defining a grammar and a method for evaluating generated programs.
3. Adjusting the Probe synthesiser to learn the programs from rewards.
4. Finding ways to increase exploration and analysing their effect on runtime.

The first three steps of the project were conducted together with Nicolae Filat and Timur Mukminov.

3.1 Generalised Probe

As the first step, we generalised the Probe algorithm. The pseudocode for this algorithm can be seen in algorithm 2. The generalised version makes it possible to use a different algorithm for searching the program space instead of guided bottom-up search. It also allows for easily changing the selection and update functions, and the number of programs to iterate in one cycle. The generalisation makes it easy to experiment with different parameters of the Probe algorithm.

The algorithms for searching the program space are implemented as iterators, which means that they return all the programs they enumerate one-by-one, instead of just returning the correct program once they find it. As such, the logic that keeps track of a cycle is moved from the guided bottom-up search algorithm to the Probe algorithm. The number of programs generated is used as the length of a cycle instead of the number of levels because a generic program iterator might not iterate programs by levels. A generic iterator might also not have an evaluation cache and a list of partial solutions, therefore these are also added to the Probe algorithm. However, this means that the generated programs have to be evaluated in the Probe algorithm. When using the generalised Probe algorithm with a program iterator that needs to evaluate the program itself, such as guided bottom-up search, this can result in double evaluation. This can be avoided by having the iterator return the result along with the program. The pseudocode in algorithm 2 does not include this optimisation.

3.2 Defining and Evaluating MineRL Programs

We defined our programs to be lists of $(steps, action)$ instructions, where $steps$ indicates the number of steps to take with the given $action$. Having the ability to take multiple steps with a single instruction significantly reduces the number of programs that have to be generated for doing the same action multiple times in a row. The evaluation function iterates over the list of instructions and executes each $action$ $steps$ number of times in the environment.

The environment must be reset between evaluations. Since fully resetting the environment takes a lot of time, we instead opted for teleporting the agent back to the start. This works well in the case of navigation environments, but might not work for other types of environments.

3.3 Program Synthesis from Rewards

Since we were unable to obtain the datasets for the MineRL environments, we could not use traces to guide the search. Instead, we used only the reward returned from the environment. To do that, we keep track of the highest reward achieved. After running the program iterator for a number of iterations, if a program with a higher reward was found, we update the grammar to use the program with the highest reward for the first instructions and restart the iterator from the beginning. This way, the agent continues from the program with the highest reward for all subsequent programs.

To make synthesising programs in this way feasible, we used the dense navigation environments, which return a reward after each step based on how much closer the agent gets to the goal. For example, moving half a block closer to the goal would yield a reward of 0.5.

However, this method does not work if the agent is not able to reach the goal from the place with the highest reward. This can happen if there is a hole on the way to the goal from which the agent cannot get out, or there is a wall directly in the direction of the goal.

3.4 Probe with Rewards

In order to make Probe work with rewards, we had to make some changes. Firstly, the algorithm no longer takes a set of examples as input, as it only learns from rewards. We keep track of the best reward and the program which obtained that reward. Because we no longer have examples, we cannot use them to determine partial solutions. Instead, we define a partial solution to be a program that improves the best reward. We also had to redefine observational equivalence. Because we were using the navigation environments, we decided that two programs are observationally equivalent if they end up in approximately the same position. The x- and z-coordinates are rounded to one digit, and the y-coordinate (altitude) is floored.

Algorithm 2: Generalised Probe algorithm

Input: PCFG $\mathcal{G}_p$, set of input-output examples $\mathcal{E}$, program iterator I, selection function SELECT, update function UPDATE, max. time T, cycle length C
Output: A solution P or $\perp$

function PROBEGENERALISED($\mathcal{G}_p$, $\mathcal{E}$, I, SELECT, UPDATE, T, C)
  $E \leftarrow \emptyset$ ▷ Initialise evaluation cache
  $S \leftarrow \emptyset$ ▷ Initialise program iterator state
  while $time < T$ **do**
    $i \leftarrow 0$
    $PSol \leftarrow \emptyset$ ▷ Initialise partial solutions
    while $i < C$ **do** ▷ Generate C programs (one cycle)
      $P \leftarrow \text{ITERATE}(I, S)$ ▷ Get next program
      $R \leftarrow \text{EVAL}(P, \mathcal{E})$ ▷ Evaluate program
      if $R = \mathcal{E}$ **then**
        return P ▷ Solution found
      if $R \in E$ **then**
        continue ▷ Result in evaluation cache
      if $R \cap \mathcal{E} \neq \emptyset$ **then** ▷ Program solves some examples
        $PSol \leftarrow PSol \cup P$ ▷ Add program to partial solutions
      $E \leftarrow E \cup R$ ▷ Add result to evaluation cache
    $PSol \leftarrow \text{SELECT}(PSol)$ ▷ Select promising partial solutions
    if $PSol \neq \emptyset$ **then** ▷ Promising partial solutions found
      $\mathcal{G}_p \leftarrow \text{UPDATE}(\mathcal{G}_p, PSol, E)$ ▷ Update grammar
      $E \leftarrow \emptyset$ ▷ Reset evaluation cache
      $S \leftarrow \emptyset$ ▷ Reset program iterator state
  return $\perp$

The selection function chooses five programs with the highest reward from the partial solutions. The grammar is then updated using the selected programs. The program with the highest reward is used to update the grammar such that all new programs start with the actions from this program. This way, all subsequent programs start searching from the position of the best program. Because the start of the program is fixed to the actions of the best program, only the last action of the promising partial solutions is used to update the probabilities of the grammar. The probabilities of each rule R are updated according to the formula:

$$p(R) = \frac{p_p(R)^{1-FIT}}{Z}$$

where

$$FIT = \min(\max_{\{P \in PSol | R \in tr(last(P))\}} \frac{r(P)}{100}, 1)$$

where p_p is the previous probability of R, $last(P)$ is the last action of program P, and $r(P)$ is the reward obtained by P.

4 Experiments and Results

This section describes the experiments conducted in order to increase the amount of exploration. It gives the parameters used to run the experiments, and the results of these experiments.

4.1 Setup

The code was implemented in Julia using the Herb.jl library (Hinnerichs and Dumančić 2024). The code is available on the `probe-with-minerl-explore` branch of the HerbSearch GitHub repository[1]. The instructions on how to run the experiments are available on the same branch in the `experiment_setup.md` file.

Version 0.4.4 of MineRL was used for the experiments. All the experiments use a modified version of the `MineRLNavigateDense-v0` environment. In this environment, the agent has to reach a diamond block approximately 64 blocks away from the spawn point. The agent is given a reward after each step based on how much closer it gets to the goal, and a reward of 100 upon reaching the goal. The modified environment, `MineRLNavigateDenseProgSynth-v0`, adds the ability to send chat commands. It also adds the position of the agent to the observation space. These changes are used for teleporting the player to the spawn point instead of hard resetting the environment between evaluations. The chat commands are also used to give the agent infinite health and food, and to disable mobs. In the original environment, the compass that points to the goal has a small randomised offset from the goal. Because the compass is used to calculate the reward, this randomisation has been turned off in order to get an accurate reward based on the actual distance to the goal. The modified version can be found on the `prog-synth` branch of the forked MineRL GitHub repository[2].

All the experiments were run on a Lenovo Legion 5

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

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

$S \rightarrow$ `[`A`]`
$A \rightarrow$ `(`T`, Dict("move" =>` D`, "sprint" => 1, "jump" => 1)`
$D \rightarrow$ `forward` | `back` | `left` | `right` | `forward-left` | `forward-right` | `back-left` | `back-right`
$T \rightarrow$ `5` | `10` | `25` | `50` | `75` | `100`

Figure 1: Grammar of the first experiment.

15ARH05 laptop running Arch Linux. This laptop has an AMD Ryzen 7 4800H processor, and 16 GB of RAM. The experiments were run ten times with a 20-minute timeout per run. The low number of runs and high timeout are due to the fact that evaluating a program takes a long time because all the actions have to be simulated in the environment. The experiments were run with the following world seeds: 6354, 95812, 958129, 999999, and 11248956. These seeds are used to generate the Minecraft worlds. In the following section, the worlds generated from these seeds are also referred to as worlds 1, 2, 3, 4, and 5, respectively.

4.2 Experiments

The experiments described in this section are additive, meaning that the parameters that are not explicitly changed in experiment N are the same as in experiment $N-1$.

Experiment 1 The first experiment uses the algorithm described in section 3.4 with cycle length set to six. The grammar used is shown in figure 1, where S is the starting symbol. As described in section 3.4, the first rule gets replaced with $S \rightarrow$ `[best_program;` A`]` in the update function. This first experiment is quite exploitative, as it continues searching only from the best program and allows for only one action after the best program. This serves as a baseline for making the algorithm more explorative.

Looking at the runtimes of the experiment in figure 2, we can see that it is able to find a solution for worlds 1, 3, and 4 in less than 600 seconds. However, because this experiment is too exploitative, it gets stuck in worlds 2, and 5. In world 2, there is a hole between the agent and the goal. The agent goes into the hole and receives a higher reward than previous programs, making subsequent programs continue from the hole. Because the hole is deep, there is no way to get out of it. In world 5, there is a cave in front of the agent and the goal is in the direction of the cave, but slightly to the left. Because of the way programs are enumerated, the agent first goes forwards into the cave. It receives a higher reward, and is unable to exit the cave because the reward would decrease.

Experiment 2 The second experiment changes the grammar to allow taking multiple actions after the best program. The rule $ACTS \rightarrow$ `[`A`]` | `[`$ACTS$`;` A`]` is added to the grammar, and the starting rule is changed to $S \rightarrow ACTS$. The update function now changes the first rule to $S \rightarrow$`[best_program;` $ACTS$`]`. This should allow the agent to exit the cave in world 5.

However, looking at figure 2, we can see that world 5 still timed out. The agent was unable to exit the cave, even though it could take multiple actions after the best program.

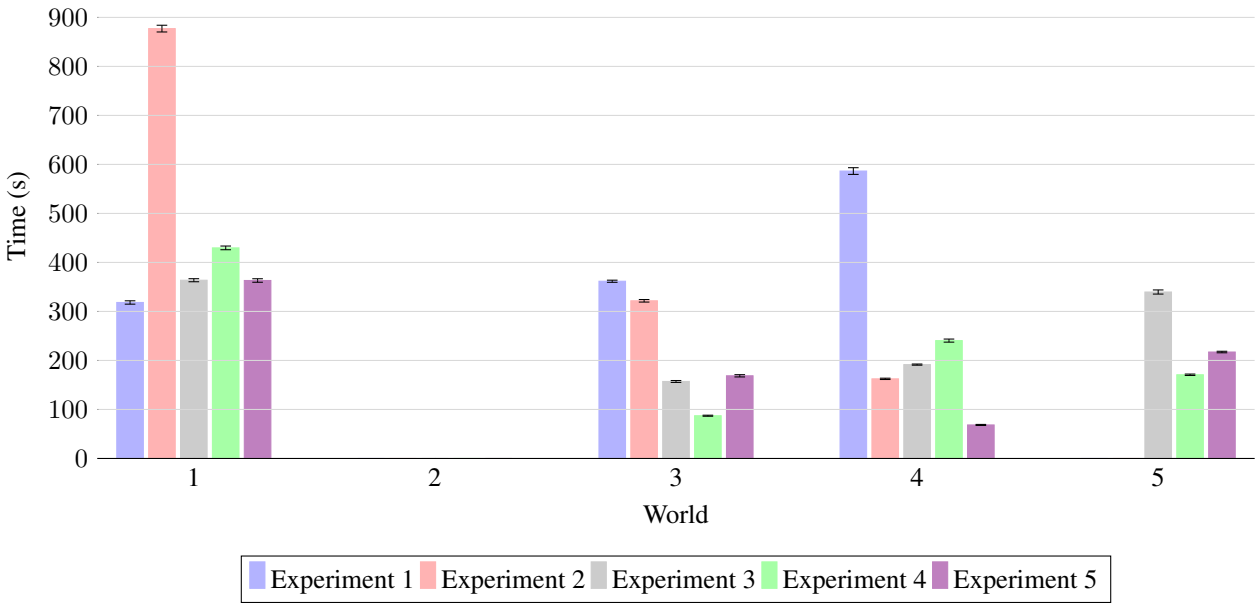

Figure 2: Average runtimes of experiments 1-5 over ten runs in seconds. A missing bar means that the experiment timed out for at least one run. The graph also shows the standard deviation of each experiment.

Because the exit is behind the agent, and the grammar has obtained higher probabilities for going forwards from previous programs, the actions that lead out of the cave have low probabilities. Getting out of the cave and increasing the reward would take many actions. Finding a program that takes many low-probability actions would take a lot of time.

Although worlds 1, 3, and 4 do not need to perform multiple actions after the best program in order to reach the goal, we can see from figure 2 that this experiment affects their runtimes. Even though they do not make use of the rule $ACTS \rightarrow \texttt{[}ACTS\texttt{;}\ A\texttt{]}$, they have to use the extra rule $ACTS \rightarrow \texttt{[}A\texttt{]}$, which adds to the costs of the programs and affects the probabilities. The rule for taking multiple actions also affects the probabilities even when it is not used. This was confirmed by running the first experiment again with the extra rule $ACTS \rightarrow \texttt{[}A\texttt{]}$, and a dummy rule, $D \rightarrow \texttt{0}$, instead of the rule for multiple actions. With these two added rules, experiments 1 and 2 have the same runtime.

Experiment 3 The order of the $(steps, action)$ tuple in the grammar of experiment 3 is changed to $(action, steps)$. This way, the different directions are enumerated before the different numbers of steps. Instead of trying all the step sizes for one direction, the synthesiser now tries all the directions with one step size. The cycle length is also changed to eight in order to enumerate all eight directions in one cycle. This should help the synthesiser find the correct direction faster.

As we can see from figure 2, this drastically improved the runtimes of worlds 1 and 3, while only slightly increasing the runtime of world 4. Worlds 1 and 3 have more obstacles, requiring direction changes, which could explain why this experiment improved their runtimes this much. This experiment is also able to solve world 5 because it does not immediately go forwards into the cave. Instead, it explores more directions and finds that going slightly left gives a better reward.

Experiment 4 Experiment 4 aims to further increase exploration by only taking into account the directions from the last best program, and not exploiting the ones from previous best programs. This is done by giving the last direction a higher probability, while giving the other directions lower, uniform probabilities. In the update function, the last direction is given a cost of 1, and other directions a cost of 10, before normalising. In case the last direction does not increase the reward, instead of exploiting directions that were used in previous best programs, it now explores all directions uniformly to find the right direction. By doing this, the synthesiser should be able to find the right direction faster in worlds that require more direction changes.

This experiment significantly improved the runtimes of worlds 3 and 5, but increased the runtimes for worlds 1 and 4.

Experiment 5 A lot of time is usually spent searching near the goal. This is partly caused by higher probabilities for higher steps sizes attained from past best programs. When the agent is close to the goal, these probabilities cause the agent to travel too far and miss the goal. This issue is remedied in experiment 5 by not updating the probabilities of step sizes, leaving them to be uniform. This way, programs are always enumerated from lowest to highest step sizes. This should also not affect the search before the end too much because, with cycle length eight, this experiment is able to cover all step sizes for one direction in a single cycle.

From figure 3 we can see that this experiment had the expected effect on world 4 by decreasing the search time near the end and keeping the rest of the search roughly the same. However, this did not work as intended for the other worlds, as it affected the search before the end too much.

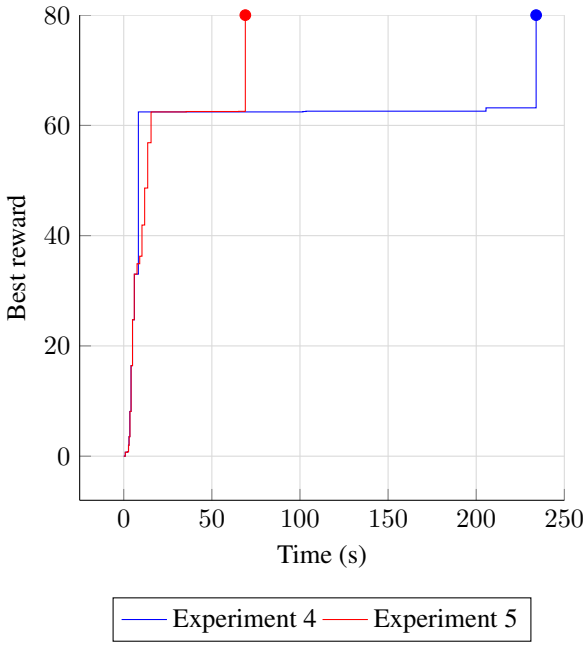

Figure 3: The best reward over time of the first run of experiments 4 and 5 in world 4.

Experiment 6 The final experiment is the most explorative – it does not exploit any of the directions or step sizes used in previous best programs. The initial costs of the grammar rules are random. In the update function, instead of updating the costs based on the last action, all the costs are again randomised. Having random costs could help the synthesiser avoid the cave in world 2. The costs are chosen randomly from $[1, 3]$ using the default random number generator in Julia, before normalising. The seed for the random number generator is the same as the world seed, and the random number generator is not reset between runs.

From table 1 we can see that experiment 6 is not able to solve any of the worlds consistently. We can also see that there is a big difference between the minimum and the maximum runtime for each world. Both of these results were expected because the directions and step sizes are random. It also seems that this experiment tends to do better in worlds with fewer obstacles, such as worlds 3 and 4. Furthermore, this experiment is the only one that was able to solve world 2, but only in two of the ten runs.

World	Successes	Min. time (s)	Max. time (s)
1	4	427	1027
2	2	264	930
3	8	178	412
4	9	44	385
5	6	317	1076

Table 1: Results of experiment 6 over ten runs. This table shows the number of successful runs, the minimum runtime, and the maximum runtime of successful runs per each world.

5 Conclusions and Future Work

The goal of this research project was to explore a novel method of specification in program synthesis – rewards. This was done by adjusting the Probe program synthesiser to synthesise programs for the dense navigation environments of MineRL. Changes were made to the Probe synthesiser to enable learning from rewards, which include redefining partial solutions, observational equivalence, and the function for updating the probabilities of grammar rules. The synthesiser was also generalised to allow for experimenting with different parameters more easily. To guide the search, the grammar was updated after a number of iterations to use the program with the best reward as the first steps for subsequent programs. Because the synthesis could get stuck in local maxima, different ways of increasing exploration were investigated to remedy that.

By changing different parameters of the Probe synthesiser, it is possible to increase exploration. The experiments done in this paper made changes to the grammar, the update function, and the number of programs enumerated in one cycle. Increasing the amount of exploration makes it possible to solve some instances that the synthesiser could not solve before, but increasing it too much could have the opposite effect. Depending on the environment, increasing exploration can either increase or decrease the runtime.

There are several paths that could be further explored in synthesising programs from rewards in MineRL:

- Using a different environment other than the dense navigation environment, such as the environment for chopping trees, which would be much more difficult as the rewards are sparse, and the action space is larger.
- Using observations from the environment in addition to the reward to guide the search, such as the RGB image, which would require image processing.
- Exploring different formulas for updating the probabilities of the rules.
- Using traces of successful solutions to guide the search.
- Using backtracking to get out of local maxima. This would require a way of finding out if the agent is stuck, which is not that straightforward.

Responsible Research

An important aspect of responsible research is reproducibility. This paper attempts to ensure reproducibility by describing the experimental setup and the conducted experiments in detail. Section 4.1 gives a thorough overview of the experimental setup, along with the specifications of the system on which the experiments were run. This section also provides a link to the GitHub repository which contains the code for the experiments and instructions on how to run these experiments, such that the results could be easily reproduced. Section 4.2 describes exactly what was changed for each experiment. Furthermore, experiments that make use of random number generators use a fixed seed for reproducibility.

References

Albarghouthi, A.; Gulwani, S.; and Kincaid, Z. 2013. Recursive Program Synthesis. In *Proceedings of the 25th International Conference on Computer Aided Verification*, 934–950. Springer.

Amiranashvili, A.; Dorka, N.; Burgard, W.; Koltun, V.; and Brox, T. 2020. Scaling Imitation Learning in Minecraft. arXiv:2007.02701.

Barke, S.; Peleg, H.; and Polikarpova, N. 2020. Just-in-Time Learning for Bottom-Up Enumerative Synthesis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA): 1–29.

Gulwani, S. 2011. Automating String Processing in Spreadsheets using Input-Output Examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 317–330. New York, NY, USA: Association for Computing Machinery.

Gvero, T.; Kuncak, V.; Kuraj, I.; and Piskac, R. 2013. Complete completion using types and weights. *ACM SIGPLAN Notices*, 48(6): 27–38.

Hinnerichs, T.; and Dumančić, S. 2024. Herb.jl: A library for defining and efficiently solving program synthesis tasks in Julia. https://github.com/Herb-AI/Herb.jl. GitHub repository.

Jha, S.; Gulwani, S.; Seshia, S. A.; and Tiwari, A. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 215–224. New York, NY, USA: Association for Computing Machinery.

Kanervisto, A.; Karttunen, J.; and Hautamäki, V. 2020. Playing Minecraft with Behavioural Cloning. arXiv:2005.03374.

Koza, J. R. 1994. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2): 87–112.

Mao, H.; Wang, C.; Hao, X.; Mao, Y.; Lu, Y.; Wu, C.; Hao, J.; Li, D.; and Tang, P. 2022. SEIHAI: A Sample-Efficient Hierarchical AI for the MineRL Competition. In *Distributed Artificial Intelligence*, 38–51. Springer.

Nguyen, H. D. T.; Qi, D.; Roychoudhury, A.; and Chandra, S. 2013. SemFix: program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, 772–781. San Francisco, CA, USA: IEEE Press.

Singh, R.; and Gulwani, S. 2016. Transforming spreadsheet data types using examples. *ACM SIGPLAN Notices*, 51(1): 343–356.

Singh, R.; Gulwani, S.; and Solar-Lezama, A. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 15–26. New York, NY, USA: Association for Computing Machinery.

Skrynnik, A.; Staroverov, A.; Aitygulov, E.; Aksenov, K.; Davydov, V.; and Panov, A. I. 2021. Hierarchical Deep Q-Network from Imperfect Demonstrations in Minecraft. *Cognitive Systems Research*, 65: 74–78.

Torlak, E.; and Bodik, R. 2013. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, 135–152. New York, NY, USA: Association for Computing Machinery.