Program Synthesis from Rewards using Probe and FrAngel

Impact of Exploration-Exploitation Configurations on Probe and FrAngel in Minecraft

Nicolae Filat[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 23, 2024

Name of the student: Nicolae Filat
Final project course: CSE3000 Research Project
Thesis committee: Sebastijan Dumancic, Tilman Hinnerichs, Wendelin Böhmer

Abstract

Program synthesis involves finding a program that meets the user intent, typically provided as input/output examples or formal mathematical specifications. This paper explores a novel specification in program synthesis - learning from rewards. We explore existing synthesizers, Probe and Frangel, to solve navigation tasks inside the popular Minecraft game. The problem formulation is inspired by reinforcement learning but was adapted to program synthesis. Similar to reinforcement learning, balancing exploration and exploitation is essential for solving the task efficiently. Excessive exploration can prevent finding the correct program because the feedback from the environment is not used. On the other hand, excessive exploitation is not ideal, as seemingly promising programs might not lead to the actual solution. This work compares different trade-offs between *exploration* and *exploitation* of two state-of-the-art algorithms, Probe and FrAngel when applied to Minecraft environments.

1 Introduction

Instead of writing code for computers, what if the computers could write code for us? The field of *program synthesis* aims to explore this exact question. If program synthesis were fully realized for any user specification, this would revolutionize many fields within computer science. In the field of software development, instead of manual coding, developers could define the desired functionality of a system, and the synthesizer will generate the code for them. This would exponentially increase the efficiency of creating software, leading to massive technology growth.

Traditionally, program synthesis relies on input-output examples to define user intent. Defining example-based specifications for programs interacting with a dynamic environment is not trivial. This paper explores a novel approach that uses rewards to define user intent. The goal is to find a program that completes a user-defined task by observing the environment's state and rewards. This could be used to generate programs in dynamic environments such as playing video games. In this work, the considered environment is the Minecraft game. The problem setup is similar to reinforcement learning, where the agent learns a decision policy by interacting with the environment. Decision policies are often learned through complex neural networks, making it difficult to reason about them. However, with program synthesis, the policy is described as a program (e.g., lines of code), which is much easier for humans to understand and modify.

Balancing exploration and exploitation is an important aspect when interacting with dynamic environments. To illustrate this idea, consider a treasure hunt in which the initial wandering to gather hints represents *exploration*, and using those hints to find the treasure represents exploitation. If one exploits too soon, they risk following misleading hints, whereas excessive *exploration* may result in time running out. Thus, balancing exploration and exploitation is the key to guiding the search effectively. Likewise, in program synthesis, balancing exploration and exploitation is essential for efficiently searching for the correct program. Figure 1 shows a visual representation of this tradeoff.

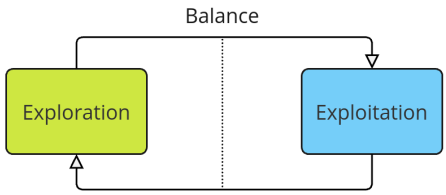

Figure 1: Visual representation of balancing exploration and exploitation.

This balance is a fundamental dilemma that arises in many domains beyond treasure hunting, including decision-making problems and machine learning tasks [1]. Achieving the desired outcome in decision-making problems requires carefully balancing the two extremes [2].

In the context of program synthesis, excessive exploration can prevent finding the correct program because the search fails to incorporate the environment's feedback. On the other hand, excessive exploitation is not ideal, as seemingly promising programs might not lead to the actual solution.

This paper explores different exploration-exploitation configurations of two existing state-of-the-art program synthesis algorithms: Probe [3] and FrAngel [4]. The two algorithms were chosen because they include exploration and exploitation phases. To understand the impact of different exploration and exploitation trade-offs, the following research questions are considered:

- *How do we use existing synthesizers to learn from rewards?*
- *How do different exploration-exploitation configurations affect the performance of FrAngel and Probe in MineRL?*

To answer the above questions, the paper makes the following contributions:

- Allow the Probe algorithm to support iterators different from the systematic bottom-up iterator.
- Enable FrAngel to work with arbitrarily domain-specific languages (it was only implemented for Java)
- Generalize Probe and FrAngel to allow for reward-based specifications
- Provide empirical results on the effectiveness of various exploration-exploitation configurations in Probe and FrAngel.

2 Related work

While recent work in program synthesis has explored defining specifications as rewards, experimental work on tradeoffs of exploration exploitation has not yet been done. The work of Natarajan et al. [5] outlines the concept of *programming by rewards*, with the goal of finding a decision function f

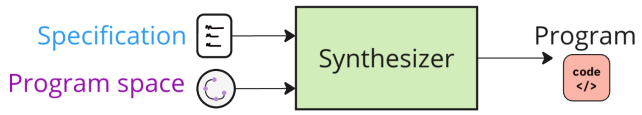

Figure 2: Visualization of the program synthesis process. Given a user specification and a program space to search in, the synthesizer finds a satisfying program.

that maximizes the expected reward. Furthermore, the study by Trivedi et al. [6] utilizes program synthesis to generate programs for reinforcement learning tasks solely based on rewards. However, the papers do not motivate the reasoning behind the choice of exploration-exploitation parameters. This research aims to address the gap in the current literature.

3 Background

This section provides the necessary background to understand the paper. Section 3.1 describes the field of program synthesis. Sections 3.2 and section 3.3 describe the concepts of bottom-up search and probabilistic context-free grammars to ease the understanding of the Probe algorithm. Finally, the MineRL experiment environment is described in 3.4.

3.1 Program synthesis

In program synthesis, the goal is to find a program that satisfies the user intent. This field is challenging due to the diversity of user specifications and the large, often infinite, program search space [7]. Existing work in program synthesis uses specifications defined as input/output examples [8], incomplete programs [9], traces [10] or even natural language [11] [12]. Given the user's intent, a program is searched to satisfy the specification. The program search space is usually defined using a grammar $\mathcal{G}$ that can either be a general-purpose programming language (e.g., Python, Java) or a domain-specific language - a language specifically designed for a class of problems. The goal is to synthesize a program $p \in L(\mathcal{G})$ that satisfies the user specification $\mathcal{E}$. A visualization of the program synthesis process is shown in Figure 2.

Common algorithms for exploring the program space include using enumeration search (e.g., BFS, DFS, A*), constraint-based solvers (e.g., SAT, SMT), stochastic search algorithms (e.g., Genetic algorithms), and machine-learning-based [11] (e.g., AlphaCode).

3.2 Bottom-up enumeration

One often used method in program synthesis is bottom-up search. Bottom-up search is a type of enumerative search that systematically explores the search space in a defined way (e.g., program depth). The procedure generates programs starting from terminal symbols and slowly builds up towards more complex programs. It uses dynamic programming to remember previously generated programs to efficiently generate new ones.

3.3 Probabilistic context-free grammars

Some synthesizers use a probabilistic context-free grammar (PCFG) to explore the program search space. A PCFG extends context-free grammar (CFG) by assigning probabilities to each production rule, enabling the grammar to guide the search process based on these probabilities. The Probe algorithm described in more detail in section 4 uses a PCFG to guide the search.

3.4 Minecraft Enviornment

MineRL [13] is a research environment based on the popular game Minecraft, designed to facilitate research in reinforcement learning and artificial intelligence in general. It provides a platform where agents can learn and perform complex tasks by interacting with a dynamic and rich 3D world. MineRL includes various predefined tasks and challenges, such as navigation, resource gathering, and crafting, which require strategic planning and problem-solving skills. The observed environment state is represented as an RGB image of the Minecraft screen. The environment provides actions that mirror those of human players (e.g., navigation, inventory management, crafting).

The MineRL environment provides many different tasks: navigating from point A to point, chopping trees, finding diamonds, etc. The reward function can either be *dense*, meaning that rewards are provided frequently (e.g., the agent is moving closer to target), or *sparse*, meaning that rewards are given infrequently, often only upon task completion (e.g., the agent has mined diamond).

4 Probe algorithm description

The Probe [3] synthesizer is an algorithm that uses a probabilistic grammar to find a program that satisfies an input-output specification. The probabilistic grammar is updated on the fly based on programs that partially satisfy the specification. The insight used is that such programs often share syntactic similarities with the final solution . The bottom-up search enumerates programs from the probabilistic grammar in order of decreasing likelihood (highest probabilities first). On a high level, the algorithm is a bottom-up search guided by a probabilistic grammar that favors programs that partially satisfy the specification (i.e., some tests pass).

To speed up the search process, the algorithm checks if a program's execution output matches that of a previous program. If that is the case, the two programs are considered equivalent, and the second program is excluded from further consideration. This technique is called *observational equivalence* and is a powerful pruning technique employed in program-synthesis [14].

The number of cycles the bottom-up iterator completes before updating the grammar is an important setting for balancing exploration and exploitation. A low setting favors exploiting the grammar probabilities more often, while a high setting favors exploring new programs. Finding a suitable value for the cycle length is important for an effective balance of exploration and exploitation. We believe that no universal cycle length value fits all problem configurations; instead, the appropriate value should be investigated through experimentation.

```
// #1. (10, 9) -> []
// #2. (10, 10) -> []
// #3. (10, 11) -> [10]
// #4. (10, 12) -> [10, 11]
// #5. (-2, 2) -> [-2, -1, 0, 1]
List<Integer> getRange(int start, int end) {
 ArrayList<Integer> list = new ArrayList<>();
 for (int i = 0; start + i < end; i++)
   list.add(Integer.valueOf(start + I));
  return list;
}
```

Listing 1: Example of program synthesized by FrAngel with 5 input/output examples given in comments. This example was taken from the original FrAngel paper.

5 FrAngel algorithm description

FrAngel [4] is a component-based synthesis algorithm that synthesizes Java functions with complex conditionals. FrAngel uses input-output examples and the function signature to find the correct program. The core idea of the algorithm is to use *random search* to explore programs that partially satisfy the specification and reuse them to guide the search. The algorithm *mines* useful fragments from partial solutions, which are then used to bias the random program generator towards more promising programs. This process guides the search towards exploiting partially successful programs.

FrAngel utilizes a concept called *angelic conditions* to handle conditions within the generated programs. These conditions are optimistically evaluated during the initial search phase, meaning that the most favorable outcome for the condition is assumed. Once a potential solution is identified, these conditions are refined and validated to ensure the final program handles all conditions correctly.

5.1 Example synthesis

As an example of the capabilities of the FrAngel synthesizer, consider the code example in 1 that was taken from the original paper [4]. In this task, the synthesizer is asked to write a function `getRange(int start, int end)` that returns the number in the interval `[start, end)`. Besides the function signature, input-output examples are provided as input to the algorithm. FrAngel deduces from the function signature that it can use `List`, `Integer`, and all their superclasses within the function body. Additionally, the user specifies that the `ArrayList` class is also permitted. Despite the large search space, FrAngel finds the correct solution in seconds. This example proves the efficiency of FrAngel in synthesizing complex Java programs.

5.2 Fragment mining

Similar to Probe, FrAngel uses the insight that partially successful programs often share syntactic similarities with the complete solution. To leverage this insight, FrAngel extracts subprograms from the found partials solution called *fragments*. The random program generation procedure works by recursively selecting grammar rules to construct a program. It begins by selecting a rule, which could be a statement, expression, or a constant. If the rule has children (e.g., the body of a loop or branches of a conditional statement), the procedure calls itself recursively to generate the child components. This recursive process continues until all terminal components are created. At each recursive step, based on `use_fragment_probability`, the procedure chooses to either use a fragment or generate a random child from scratch. If fragments are to be used, they can either be used *entirely* (i.e., copy-paste), or random modifications can be made. With probability `use_entire_fragment_probability` fragments are used; otherwise, random modifications are made to the fragment using `mutation_probability`.

5.3 Configuration variables for exploration-exploitation

There are a few configuration variables in the fragment mining generation to tune exploration and exploitation:

- `use_fragments_chance` – A value of 0 indicates random search, meaning constant exploration, while a value of 1 represents continuous exploitation of previously mined fragments (if any). The original paper uses a value of 0.5 to have an equal trade-off between random search and mining fragments.
- `use_entire_fragment_chance` – A value of 0 implies modifications can be made to fragments when inserted, whereas a value of 1 indicates that fragments never use random mutations. Depending on the structure of the fragments and the problem type, random mutations might help. However, mutating fragments too often could lead to excessive exploration and misuse of partially correct programs.

6 Methodology

The main contributions of this paper are:

1. Generalize Probe to allow for iterators other than the bottom-up search.
2. Modify the Probe algorithm to learn from rewards instead of examples.
3. Generalize FrAngel to allow arbitrary grammars and use different iterators.
4. Use FrAngel with reward-based specifications.
5. Experiment with different exploration-exploitation trade-offs for Probe and MineRL within MineRL environments.

Steps 1 and 2 were conducted together with Nils Marten Mikk and Timur Mukminov while steps 3 and 4 were implemented by George Latsev and Alperen Guncan.

6.1 Program synthesis from rewards

A formal description is provided to understand the extension of both algorithms to reward-based formulations. The problem description is inspired by reinforcement learning and is adapted to program synthesis. The specification $\mathcal{E}$ is phrased as solving a user-defined task in a given environment. The

environment E is described by a finite set $\mathcal{A}$ of actions and a set of states $\mathcal{S}$. To guide the search, a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is provided by the environment. The reward function $\mathcal{R}$ indicates the reward that the agent receives when performing the action a in state s. The goal is to find a program p from the given grammar $\mathcal{G}$ that, after being executed, achieves the highest possible reward in the environment E. Formally, the goal is defined as $\max_{p \in L(\mathcal{G})} \text{reward}(p)$ where $\text{reward}(p)$ is the achieved reward when executing program p in the environment $\mathcal{E}$.

6.2 Probe generalization

The generalization of the Probe algorithm we implemented enables the usage of different iterators beyond the bottom-up iterator. This feature improves the flexibility of the algorithm but also enables the execution of more creative experiments. Using this generalization, an iterator that *alternates* between random search and systematic bottom-up search was implemented. The alternating iterator generates a random program with probability p, and with probability $1 - p$, it uses the systematic bottom-up iterator. The alternating iterator was used to allow for a greater degree of exploration in the Probe algorithm.

6.3 Probe learning from rewards

There are a few questions that need to be answered when generalizing the implementation to allow reward-based specifications:

- How to define partial solutions? (e.g. When does a program qualify as partially satisfying the specification?")
- How to define observational equivalence with rewards?
- How to update the grammar based on partial solutions?

Firstly, there are multiple options to consider when defining a partial solution. One option could be to define a partial solution as a program that achieves a reward r with $r > 0$. However, this is not viable since many programs can achieve positive rewards. Ideally, the set of partial solutions is small and guides the search towards solving the task. We have defined a partial solution as a program with a reward higher than the previously best achieved reward. Using this approach, the considered partial solutions will always perform better than previous programs and thus guide the search toward achieving higher rewards.

Secondly, the environment state can be used to define the observational equivalence of programs. If the output of two programs results in the same environment state s, they are considered equivalent. The state definition is specific to each environment E.

Thirdly, the probabilistic grammar needs to be updated after each synthesis cycle to bias the found partial solutions. The formula used in the original Probe paper updates the probability of rules in terms of the computed *fitness* for each rule. The fitness of a grammar rule indicates how well that rule performs in partial solutions that use it. Since Probe uses input-output examples, the fitness is computed as the highest number of input-output examples that partial solutions using that rule satisfy. The insight used is that grammar rules used in partial solutions that satisfy a large subset of the specification are more likely to appear in programs that satisfy the entire specification. In our approach, we leverage the same idea but use the ratio of the achieved reward to the best possible reward instead of the ratio of correct solved examples to a total number of examples. The best reward is a constant specific to each environment and represents the maximum achievable reward.

6.4 FrAngel generalization

This section describes the improvement made to the FrAngel algorithm to broaden its applications for synthesizing code beyond the Java programming language. The generalization of the algorithm also enables other iterators to make use of fragment mining and angelic conditions.

Arbitrary grammar definition

A major limitation of FrAngel is that it can only generate programs for the Java programming language. Given its effectiveness when used for Java programs, the algorithm is expected to generalize well to other programming languages or domain-specific languages. For this reason, we have reimplemented the algorithm to allow for arbitrary grammars as input. While most algorithm steps remain independent of the grammar choice, the choice becomes important when using angelic conditions. Identifying which grammar rule might represent a conditional statement is challenging, especially for user-defined languages. For this reason, we require the user to specify which grammar rules actually represent conditionals. This addition broadens the algorithm's application, allowing it to synthesize code for other languages, such as Python or user-defined languages.

Learning from rewards

To adapt the FrAngel algorithm to work with reward-based specifications, the reward space was discretized into evenly spaced thresholds to form test cases. Each test case checks whether the program achieves a reward higher than the respective test case threshold. This approach reformulates a reward-based formulation in terms of test cases to allow using FrAngel on reward-based specifications.

Generalize the iterator

FrAngel uses random search as its primary iteration method. However, this is not a requirement, as alternative iterators can also be used. This generalization enables iterators other than random search to use the core FrAngel ideas: fragment mining and angelic conditions. Additionally, iterators can receive information about mined fragments to enable novel uses of fragments. This could enable other stochastic iterators (e.g. Metropolis-Hastings, Genetic search) to utilize fragments.

Configurable probabilities

The original description of FrAngel includes numerous hard-coded probability values that impact fragment mining and angelic condition generation, therefore influencing the balance between exploration and exploitation. Our implementation introduces a modular configuration for these probabilities. This flexibility facilitated experimentation with various exploration-exploitation configurations.

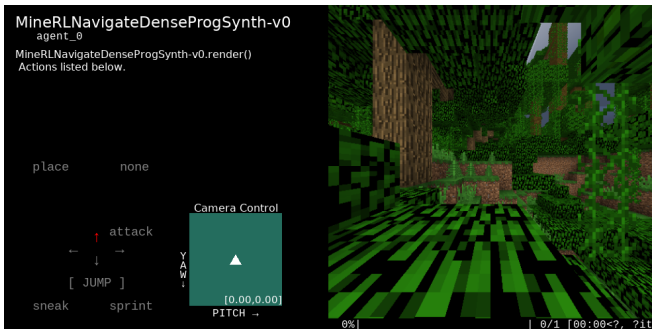

Figure 3: The starting position of the MineRL environment with seed 4231. On the left, the metadata about the environment is shown, while on the right, the view of the player is shown.

7 Experimental Setup and Results

This section outlines the experiments conducted with various exploration-exploitation trade-offs. First, the experimental setup is described in detail, followed by the presentation of results for Probe and FrAngel.

7.1 Experiment setup

The code was implemented in the Julia programming language using the Herb.jl [15] library. Instructions on how to setup the environment and run the experiments are available on Github [1].

Navigate Task

The `MineRLNavigateDense-v0` environment was used for experimentation because it provides an appropriate difficulty level for synthesizing programs from rewards only. The task is to navigate to a diamond block that is 64 blocks away from the initial position of the player. The rewards are dense, meaning that the reward is defined as how much closer the agent is to the target compared to the previous frame.

Environment simplifications

Extracting information from image frames is a challenging task that typically involves the use of machine learning. Although pre-trained machine learning models can be used to detect objects such as blocks and trees, integrating these features into the synthesis algorithm would also pose significant difficulties. Therefore, we have decided to only use the environment's rewards to guide the search process. This approach essentially treats the Minecraft environment as a *black-box* that provides rewards for each action. A few environment simplifications were made because the synthesizer does not use visual information. Firstly, the player was given infinite life and stamina to prevent the player from dying while a program was being executed. Secondly, animals were removed from the environment to prevent them from obstructing the target location. Thirdly, when simulating a Minecraft run, the player was hard-coded to jump and sprint to increase the chances of reaching the goal.

[1] https://github.com/Herb-AI/HerbSearch.jl/tree/exploration-vs-exploitation-in-frangel-and-probe/

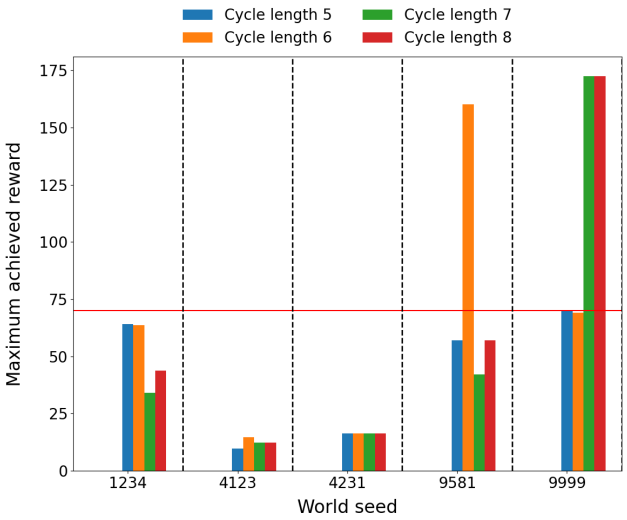

Figure 4: Plot of the maximum achieved reward when running Probe with different cycle lengths. For each seed, the cycle lengths are shown in order. The red line ($y = 70$) corresponds to a program that reaches very close to the diamond block but does not touch it. When the diamond block is touched, the environment awards an additional reward of 100. Hence, any bar above 100 indicates that the navigation task was solved.

Methodology

Many navigation tasks can be created by providing a seed to the Minecraft world generation. Each world has different start and target positions. The following environment seeds were chosen: $1234, 4123, 4231, 9581, 9999$ because they generate worlds with different difficulty levels (e.g., hills, forests, water, sand). As a visual representation of a potential world configuration, Figure 3 shows the starting position of the player for environment 4231.

The experiments were run on an Aspire 7 laptop running Ubuntu with AMD Ryzen 7 5700U processor and 16GB of RAM. All experiments involved running different configurations of exploration-exploitation for the `MineRLNavigateDense-v0`. To account for randomness, each configuration was run 3 times per world seed, and the average and standard deviation of the runtimes for each seed were plotted. The reason behind the low number of runs is that the MineRL simulation environment is slow, and it was not feasible to run more experiments within the time limit.

7.2 Probe Results

Grammar definition

The grammar used by the Probe algorithm to synthesize programs for the MineRL environment is shown in Figure 5. The starting point of the grammar is the Seq symbol, which represents a sequence of actions that the program can take. Each action is defined as a direction and a multiplier that indicates how many times to go in that direction. Eight directions are considered: forward, back, left, right, forward-right, forward-left, back-left, and back-right.

Because of the increased difficulty of synthesizing a program that reaches the end goal from the beginning,

$\text{Seq} \rightarrow \text{best}, \text{M}$
$\text{best} \rightarrow \emptyset$
$\text{M} \rightarrow \text{M}, \text{A} \mid \text{M} \rightarrow \text{A}$
$\text{A} \rightarrow (\text{T}, \text{Dict}(\texttt{move} \rightarrow \text{D}))$
$\text{D} \rightarrow \texttt{forward} \mid \texttt{back} \mid \texttt{left} \mid \texttt{right} \mid \texttt{forward-left}$
$\text{D} \rightarrow \texttt{forward-right} \mid \texttt{back-left} \mid \texttt{back-right}$
$\text{T} \rightarrow 5 \mid 10 \mid 25 \mid 50 \mid 75 \mid 100$

Figure 5: The grammar definition used by the Probe algorithm in the MineRL environment

checkpointing is used. After each synthesis cycle, the program that achieves the best reward is saved as a checkpoint in the grammar using the $best$ rule. Later generated programs will include the actions of the best program and start exploring from the saved checkpoint. The checkpointing technique assumes that reaching the target goal starting from the last checkpoint is easier than starting from the initial position.

Experiments

Figure 4 shows a plot with the maximum achieved reward when running using different cycle lengths. The graph indicates that cycle lengths of 5 and 6 achieve the best performance. Both configurations get very close to the target but do not manage to touch it in the first and last world. Additionally, in the world seed 9581, the configuration with cycle length 6 succeeds in completing the task. We believe that cycle lengths with lower values perform better because they exploit partial programs more often. In every synthesis cycle, the closest position to the target is saved, and future programs start from there. Faster synthesis cycles imply faster use of checkpointing and, thus, more efficient navigation.

However, Probe's performance is relatively poor, solving only 2 out of 5 tasks within the time limit. This could be attributed to its use of the bottom-up iterator. The difficulty of the navigation task lies in finding the correct direction to move in as fast as possible. Because the systematic bottom-up iterator explores each direction sequentially, finding the correct direction of movement is inefficient. In world seed 4231, all Probe runs get stuck in a forest dead-end, from which it takes too much time to get out. Random search might fix this issue by allowing the algorithm to *"guess"* a correct direction sooner, but also to escape local maximum. The hypothesis is that the efficiency could improve if a percentage p of the generated programs is random.

To investigate this idea, experiments were run using an alternating iterator with probabilities p to $0.3, 0.5$, and 1 as shown in Figure 6. The plot shows the average runtime for solving the same world seeds as before. It can be seen the first world seed is solved compared to the previous experiment. Based on the average runtime and frequency of appearing in the graph, the best alternating random configuration is with $p = 0.3$. Configurations with $p = 0.5$ and $p = 1$ decrease the performance due to excessive exploration. This experiment suggests that using random search for 30% of the time provides a good balance between exploration and exploitation.

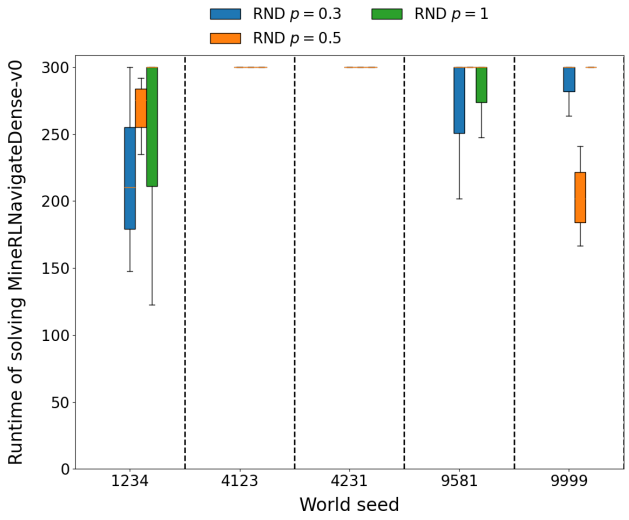

Figure 6: Box plot of the runtime solve time when running Probe with different alternating random probabilities p. The time limit is 300 seconds. When a run does not manage to solve the task, the average runtime is 300 seconds, and a small yellow line is shown.

The downside of adding random exploration is that the standard deviation increases. The same configuration can solve the task one time but fail next time. Out of the 3 runs on world seed 9999 for the configuration with $p = 0.3$, at least one of them has timed out as the blue bar touches the $y = 300$ line. However, when running without randomness, the number of solved attempts is consistent: either all the runs lead to the solution, or no run leads to the solution.

7.3 FrAngel results

The grammar used for integrating FrAngel with the Minecraft environment is similar to that used in Figure 5. To utilize the strengths of FrAngel, the grammar was extended to allow for while loops and conditions. The number of iterations a while loop makes is restricted to prevent infinite looping programs. All the experiments with FrAngel had a timeout of 200 seconds.

Figure 7 shows the mean, maximum, and minimum runtime values when running FrAngel with different mutation probabilities. It can be observed that the configuration with no mutation (i.e., `mutation_prob` = 0) generally performs better than other configurations across all world seeds. The exception is seed 4231, where the configuration with a mutation probability of 0.5 yields the best performance, being the only one to solve the seed. This environment, represented by this seed, is a dense forest with many trees, making navigation challenging. Hence, allowing more exploration with a higher mutation probability improves the performance in this case. However, this is the only environment seed where increased mutation proves more efficient.

Using fragments less often proves to be more efficient on the first 3 world seeds, as can be seen by Figure 8. The configurations with lower probabilities 0.2 and 0.4 have a lower average solve time than those with probabilities 0.6 and 0.8. This could be because the structure of the Minecraft navi-

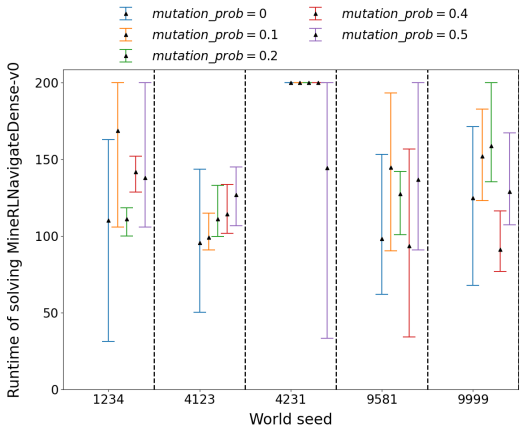

Figure 7: Plot with the maximum, minimum, and mean runtime of FrAngel when using different mutation probabilities and a constant probability of 0.4 for using fragments. The timeout is 200 seconds.

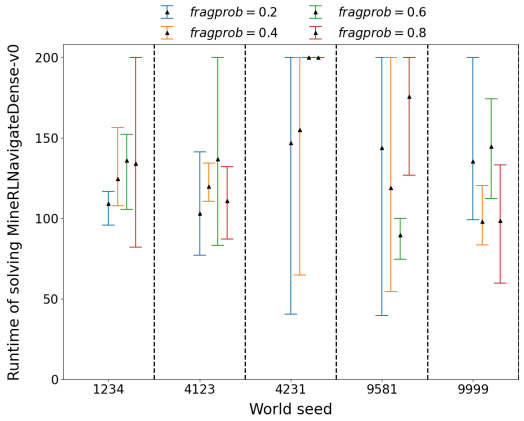

Figure 8: Plot that shows the maximum, minimum, and mean runtime of FrAngel when using different `use_entire_fragment_probabilities` and a constant mutation probability of 0.25. The timeout is 200 seconds.

gation task does not benefit from reusing fragments. If the Minecraft world is similar to a maze, using fragments is not effective; however, if the land is mostly flat and there are few obstacles, reusing fragments proves more effective. As the worlds described by seeds 9581 and 9999 are mostly flat and predictable, configurations with probabilities 0.6 and 0.8 achieve better performance. Interestingly, on seed 9581, the configuration with a probability of 0.8 performs much worse than the one with 0.6, suggesting that exploiting fragments too early may cause the player to get stuck near obstacles, such as trees, preventing further progress.

8 Responsible Research

Conducting research on learning exploration and exploitation trade-offs in program synthesis has few ethical implications. However, ensuring that the presented results are reproducible by other researchers is crucial. For that purpose, all the implementation details and the experiment data are available on Github[2] as part of the Herb framework. The Github wiki documentation contains detailed instructions on how to set up the environment and install the project dependencies. Additionally, the random number generator used was given a fixed seed to ensure the reproducibility of experiments.

9 Discussion and Limitations

It can be seen that combining random search with the vanilla algorithm increases its performance. We believe this is the case because the Minecraft navigation task requires more exploration of different movements than a systematic search. The bottom-up iterator of grammar used in experiments (Figure 5) first iterates the multiplier and then different directions meaning (5,`"forward"`), (10,`"forward"`), ...(100, `"forward"`), (5, `"back"`), (10, `"back"`)) .... Changing the grammar to iterate first over direction and over multipliers could improve the performance of the Probe algorithm.

Most of the time, the synthesizer struggles to avoid obstacles in the environment. Lacking information about nearby objects, the algorithms frequently attempt to move into trees or blocks. This issue is particularly seen with Probe. If moving forward increases the reward, Probe is more likely to reuse that direction. However, if a tree is directly in front of the player, Probe will continue to generate programs that attempt to move forward. Only after all directions that go forward have been exhausted will the algorithm try to turn right or left. This issue could be resolved if image processing is incorporated into the algorithm.

During experimentation, we noticed that the chosen random seed influences the outcome of the run. Sometimes changing the seed could make the difference between solving the task or not. Because of time constraints, each configuration was run only 3 times for each world seed. Future research should aim to run experiments for longer periods of time and on larger sets of seeds to further reduce the impact of randomness.

10 Conclusions and Future Work

This work aimed to explore using existing synthesis algorithms for reward-based specifications and investigate how different exploration-exploitation configurations affect the performance of FrAngel and Probe. The implementation of both algorithms had to be generalized to enable learning from rewards. For Probe, the changes included redefining partial solutions, observational equivalence, and the function that updates the grammar rules. For FrAngel, the algorithm was generalized to allow arbitrary grammars, and the reward space was discretized to enable learning from rewards. Furthermore, both generalizations allow for using different iterators other than the ones included in the original papers.

Both algorithms proved effective when applied to Minecraft navigation tasks. By modifying different configuration variables of the two algorithms, experiments were conducted with different exploration-exploitation trade-offs. For Probe, experiments were conducted with different cycle

[2]https://github.com/Herb-AI/HerbSearch.jl/tree/exploration-vs-exploitation-in-frangel-and-probe/

lengths and various probabilities for the alternating iterator. For FrAngel, the experiments involved changing the fragment mutation and usage probabilities to explore different exploration-exploitation strategies. In both algorithms, increased exploration appears to enhance the performance of Minecraft navigation tasks. Since the algorithms lack awareness of the environment's structure and rely only on rewards, avoiding obstacles is challenging. Hence, more exploration often results in better performance.

There are several ideas that can be further researched when synthesizing programs from rewards using MineRL:

- Use traces of human players to guide the search process.
- Integrate the visual feedback from the environment into the algorithm's decision process.
- Use random exploration or backtrack to the previous checkpoint when the reward cannot improve because the agent is stuck. This improvement could allow programs to navigate the environment more efficiently.
- Enhance the agent's ability to reason about directions. For instance, if moving right increases the reward, there is no reason to generate programs that move in the opposite direction.

References

[1] O. Berger-Tal, J. Nathan, E. Meron, and D. Saltz, "The exploration-exploitation dilemma: a multidisciplinary framework," *PloS one*, vol. 9, no. 4, p. e95693, 2014.

[2] G. E. Flaspohler, "Balancing exploration and exploitation: Task-targeted exploration for scientific decision-making," Ph.D. dissertation, Massachusetts Institute of Technology, 2022.

[3] S. Barke, H. Peleg, and N. Polikarpova, "Just-in-time learning for bottom-up enumerative synthesis," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, p. 1–29, Nov. 2020. [Online]. Available: http://dx.doi.org/10.1145/3428295

[4] K. Shi, J. Steinhardt, and P. Liang, "Frangel: component-based synthesis with control structures," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, p. 1–29, Jan. 2019. [Online]. Available: http://dx.doi.org/10.1145/3290386

[5] N. Natarajan, A. Karthikeyan, P. Jain, I. Radicek, S. Rajamani, S. Gulwani, and J. Gehrke, "Programming by rewards," *arXiv preprint arXiv:2007.06835*, 2020.

[6] D. Trivedi, J. Zhang, S.-H. Sun, and J. J. Lim, "Learning to synthesize programs as interpretable and generalizable policies," in *Advances in Neural Information Processing Systems*, vol. 34, 2021. [Online]. Available: https://arxiv.org/pdf/2108.13643.pdf

[7] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 3–4, 2017.

[8] S. Gulwani, "Programming by examples: Applications, algorithms, and ambiguity resolution," in *Automated Reasoning*, N. Olivetti and A. Tiwari, Eds. Cham: Springer International Publishing, 2016, pp. 9–14.

[9] A. Solar-Lezama, *Program synthesis by sketching*. University of California, Berkeley, 2008.

[10] K. T. Yessenov, "Program synthesis from execution traces and demonstrations," Ph.D. dissertation, Massachusetts Institute of Technology, 2016.

[11] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[12] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy, "Program synthesis using natural language," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 345–356. [Online]. Available: https://doi.org/10.1145/2884781.2884786

[13] W. H. Guss, C. Codel, K. Hofmann, B. Houghton, N. Kuno, S. Milani, S. Mohanty, D. P. Liebana, R. Salakhutdinov, N. Topin, M. Veloso, and P. Wang, "The minerl 2019 competition on sample efficient reinforcement learning using human priors," 2021.

[14] C. Smith and A. Albarghouthi, "Program synthesis with equivalence reduction," in *Verification, Model Checking, and Abstract Interpretation*, C. Enea and R. Piskac, Eds. Cham: Springer International Publishing, 2019, pp. 24–47.

[15] T. Hinnerichs and S. Dumancic, "Herb.jl: A library for defining and efficiently solving program synthesis tasks in julia," 2024, gitHub repository. [Online]. Available: https://github.com/Herb-AI/Herb.jl