Playing Minecraft with Program Synthesis

Adapting FrAngel to Uncover Diverse Subprograms

Georgi Latsev[1]
Supervisor(s): Sebastijan Dumančić[1]**, Tilman Hinnerichs**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

Abstract

Program synthesis remains largely unexplored in the context of playing games, where exploration and exploitation are crucial for solving tasks within complex environments. FrAngel is a program synthesis algorithm that addresses both of these aspects with its fragments used for the generation of new candidate programs. We introduce a generalised version of the FrAngel program synthesis algorithm that accepts an arbitrary iterator and grammar, enabling flexible modifications. We then formulate Program by Example (PBE) program synthesis from rewards and apply this framework to Minecraft's navigation task, where dense rewards guide the algorithm's exploration. Then, we use the generalised version of the algorithm to conduct experiments in the context of exploration of the algorithm. The experiments show the importance of exploration as a step required for the exploitation and finding the wanted solution to the task while also revealing that sometimes more exploration does not necessarily mean reaching the solution faster.

1 Introduction

With the advancements in artificial agents for games such as Go [1] and StarCraft [2], the field of artificial intelligence has been increasingly interested in the search for autonomous agents capable of learning and adapting to complex game environments [3]. Games are particularly interesting as they are perfect experimenting platforms, providing environments that can be compared to real-world scenarios [4]. However, the application of program synthesis – a method that can generate various programs to solve different problems – remains largely unexplored in the context of games. There are still open questions on how to model the exploration, exploitation and decision-making in complex environments based on rewards to the typical program synthesis problem specification.

Program synthesis [5] is the process of automatically generating programs satisfying a given specification. It is applied in various fields with examples such as FlashFill in Microsoft Excel [6], allowing fast string manipulation based on user intent, automating transformations on tree-structured data [5] and synthesising code snippets to assist developers in live programming environments [7]. Various techniques for program synthesis are based on how they search the enormous space of possible programs and the program specifications they require. This paper focuses on the FrAngel algorithm, a programming by-example (PBE) algorithm, where programs are generated based on input-output examples.

FrAngel [8] is a PBE program synthesis algorithm that relies on the idea that larger programs are created from smaller, useful ones. The algorithm is based on two main concepts - fragments and angelic conditions. Fragments combine exploration and exploitation by reusing parts of programs that partially follow the specification to generate new programs that align with the desired behaviour. On the other hand, angelic conditions are placeholders for the conditions of control structures, such as ifs and loops.

Both exploration and exploitation are crucial for the algorithm's performance. Exploration discovers new useful fragments and programs from the unexplored search space, while exploitation reuses these fragments to guide the search towards the solution. The balance between them ensures that the algorithm efficiently finds the desired programs by leveraging known successes while continually searching for new potential solutions.

This paper intends to explore the capabilities in terms of exploration of the FrAngel program synthesizer and bridge the gap in the application of program synthesis in the context of playing games with rewards, in particular Minecraft, a survival sandbox game.

The main question is how to adjust the FrAngel program synthesizer to discover diverse subprograms in gaming contexts like Minecraft that can then be exploited to reach the solution faster. The paper outlines program synthesis from rewards and how to discover useful actions in game environments. It explores various modifications for exploration in the FrAngel algorithm and their impact on the overall performance. Different variants for defining the simpler output examples and their effect on initial fragments and the subsequent iterations are tested. It tries an alternative approach for generating initial programs, prioritising less frequently used basic rules so far to accelerate the exploration of possibly useful ones. Additionally, it examines the impact of limiting the large fragment symbols to encourage the exploration of different combinations of fragments and randomly generated rules. Lastly, the paper explores different parameter configurations to adjust the amount of exploration in the second phase of the FrAngel algorithm, where there are already some fragments and both exploration and exploitation are present.

This paper explains a generalised implementation of the FrAngel algorithm and models the reward problem as checkpoints represented in the input-output examples.

The structure of this paper is as follows. In Section 2, the background of the research and the related work is outlined, providing the necessary context. Section 3 discusses the methodology, provides a generalised version of FrAngel and defines program synthesis from rewards. Following this, Section 4 presents the experimental setup, results, and conclusions. Next, Section 5 addresses responsible research practices, emphasising on reproducibility. Finally, Section 6 concludes the research and suggests future work.

2 Background and Related Work

Program Synthesis

Program synthesis [5] is searching the space of possible programs to find one that meets the specific requirements. There are various methods for searching leading to different program synthesis approaches, including deductive synthesis, which uses theorem provers to construct logical programs from formal specifications, and transformation-based synthesis, which iteratively transforms high-level specifications into low-level programs. More recent methods include inductive synthesis, which relies on input-output examples, and the use of genetic programming to evolve programs.

Two main challenges in the realm of program synthesis are the size of the search space and the user intent. The search space should balance expressiveness and efficiency, and often domain-specific languages/grammar are employed to make a program more manageable. The user intent can be expressed in various ways, such as logical specification, natural language, and demonstrations. There should be a balance between how easy and accessible the requirements are to be written as simpler ones, such as input-output examples, could lead to more ambiguity given a small number of examples - while on the other hand, formal specifications could often end up being as complex as the wanted program.

An example of a simple integer arithmetic program synthesis problem with input-output specification can be seen in Figure 1. The program synthesiser is given as input the specification 1a and the domain-specific language 1b and eventually should output a solution from the possible rules that passes all examples as shown in 1c.

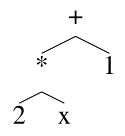
Programming by Example (PBE) is a popular inductive synthesis approach where the goal is to generate programs that meet a set of input-output examples provided by the user. PBE algorithms must balance overfitting to the given examples and generalising to unseen cases. Key challenges in PBE include handling ambiguous specifications and efficiently searching the large space of possible programs. Furthermore, many existing algorithms have difficulty in generating complex control structures, such as loops and conditionals, which limits their applicability to more sophisticated programming tasks.

$$\begin{array}{l} x = 1 \rightarrow 3 \\ x = 2 \rightarrow 5 \\ x = 3 \rightarrow 7 \\ x = 4 \rightarrow 9 \end{array}$$

(a) Input-output spec

$$\begin{array}{l} Num = 1 \mid 2 \\ Num = \mathrm{x} \\ Num = Num + Num \\ Num = Num * Num \end{array}$$

(b) Grammar rules



(c) AST for $2x{+}1$

Figure 1: Program synthesis example for simple integer arithmetic.

Programming by Rewards (PBR) [9] is a framework for synthesising and tuning decision functions. It takes input features, the output data type and a reward function, and with the use of continuous optimisation methods, it performs the synthesis. The reward function is a black-box function that assigns a reward value for the output of the decision function for the given input, while the decision function is a white-box function whose structure is being exploited for faster convergence and better accuracy. However, this framework is limited to loop-free DSL language with if-else statements. Furthermore, instead of providing a way to reuse already existing program synthesis algorithms, it provides a reinforced learning approach. Therefore, this work is not applicable for modelling the reward problem in already existing PBE algorithms.

FrAngel

FrAngel [8] is a component-based program synthesis algorithm that generates Java programs given input-output examples, a signature and libraries/components. It begins by randomly generating programs up to a particular size and then evaluating them, keeping fragments from useful ones that passed at least part of the examples. Furthermore, as it is often hard to guess the correct conditions for control structures, it places placeholders called angelic conditions that are only evaluated if the program is promising, meaning that it passes some of the tests in the best-case scenarios. The general structure of the algorithm can be seen in Algorithm 1.

Algorithm 1 FrAngel pseudocode [8]

Input: Target program signature S, set of libraries L, set of test cases C
Output: A program P that passes all tests

```
procedure FrAngel($S, L, C$)
    $R \leftarrow \emptyset$                                   ▷ A set of remembered programs
    $F \leftarrow \emptyset$                                   ▷ A set of fragments
    repeat until timeout
        $A \leftarrow RandBoolean()$
        $P \leftarrow GenRandomProgram$(S, L, F, A)             ▷ Step (1)
        $T \leftarrow GetPassedTests$(P, C)
        if $T = \emptyset$ then
            continue
        if $P$ contains angelic conditions then                ▷ Step (2)
            $P \leftarrow ResolveAngelic$(P, L, F, T, C)
            if failed to resolve conditions then
                continue
            $T \leftarrow GetPassedTests$(P, C)
        $P \leftarrow SimplifyQuick$(P, T)
        $T \leftarrow GetPassedTests$(P, C)
        if $P$ is the simplest program to pass $T$ then        ▷ Step (3)
            $R \leftarrow RememberProgram$(P, R)
            $F \leftarrow MineFragments$(R)
        if $T$ = $C$ then
            return $SimplifySlow$(P, C)
    return Failure
```

The algorithm can be divided into three steps: random program generation, angelic condition resolving, and fragment mining. First, it randomly generates a program that may include angelic conditions and previously mined fragments, either entire or mutated. Next, the algorithm resolves any angelic conditions, resulting in a program without them. Finally, it re-evaluates the program and mines useful fragments to be used in the generation of subsequent programs.

FrAngel uses random search and angelic conditions to simplify finding the required program. Angelic conditions act as placeholders for control structures conditions as shown in 2b, allowing FrAngel first to create a rough program without specific conditions. Once a satisfactory program is found, it assigns these placeholders concrete values. This approach enables FrAngel to handle complex behaviours and identify reusable fragments. Angelic conditions evaluate optimistically, choosing true or false to produce a correct output. The algorithm uses a bitstring representation to systematically explore potential execution paths, optimising the search process by prioritising simpler paths and pruning redundant ones. If an angelic program passes enough test cases, then the angelic conditions are resolved to generate a concrete program.

Fragments are mined from programs that pass some test cases to aid in generating new programs, focusing the search on previously successful components. A fragment is any complete subtree of a program's abstract syntax tree (AST) as shown in 2a. The simplest programs passing any subset of test cases are remembered and then fragments are extracted from them. The remembered programs, which may be discarded if simpler versions are found, help FrAngel focus on useful code components. The concept of special-case similarity supports this approach, where simpler (special-case) programs share meaningful fragments with the target program. This method leverages the overlap between these fragments to guide the synthesis process toward the target solution, although it has limitations when special-case similarity is absent, test cases are insufficient, or fragments are too complex or irrelevant.

(a) Fragments for program $5 + x$	(b) Angelic placeholders
$Fragment\_Num = 5 + x$	
$Fragment\_Num = 5$	$if(< angelic >)$ ...
$Fragment\_Num = x$	$while(< angelic >)$ ...

Figure 2: Illustration of fragments and angelic conditions in FrAngel.

Exploration

Exploration is the process of searching and discovering, and in the context of reinforcement learning, it involves finding actions that maximise the reward function [10]. It is a critical aspect because the reward function is often complex, especially in problems with sparse rewards. Various approaches to exploration include taking random actions, providing denser rewards, and exploring moves that could lead to new solutions.

Probe

Probe [11] is a guided bottom-up program synthesis algorithm that shares common possibilities in exploration and exploitation as FrAngel. It relies on two main ideas - guiding the search with probabilistic models and just-in-time learning. The program space is explored in decreasing probability, and these probabilities are updated on the fly based on whether the rules were present in partially successful programs, only satisfying a subset of the input-output examples. The algorithm combines the benefits of probabilistic guided search with the ability to be used on novel problems as it does not require training data.

3 Methodology

The research questions will be answered in the following steps: creating a generalised version of FrAngel for the experiments, adapting the reward-based MineRL environment to a program synthesis problem, defining an appropriate grammar for the task, and finally conducting experiments, explained in Section 4.

Generalised FrAngel Algorithm

This subsection outlines a generalisation of the FrAngel algorithm. It extends the original algorithm's scope, allowing it to handle more general problems beyond generating Java programs with a fixed initial program generation method. It supports any context-free grammar for the search space and various iterators. Additionally, it allows for changes in how the fragments are modified and mined. This generalisation, however, results in the loss of domain-specific optimisations available in the Java-focused version, such as listing usable libraries and leveraging grammar-specific knowledge. The pseudocode is detailed in Algorithm 2, with grey blocks indicating deviations from the original description of the algorithm.

Algorithm 2 Generalisation of FrAngel

Input: Grammar G, program iterator I, set of test cases C, and angelic conditions A
Output: A program P that passes all tests

```
1: procedure FrAngel($G, I, C, A$)
2:     $R \leftarrow \emptyset$                                  ▷ A set of remembered programs
3:     $F \leftarrow \emptyset$                                  ▷ A set of fragments
4:     $V \leftarrow \emptyset$                                  ▷ A set of visited programs
5:     $S \leftarrow null$                                       ▷ Initialise state for the iterator
6:     $AddAngelicRuleNode(G)$
7:     $O \leftarrow AddFragments(G)$                            ▷ Store fragment rule offsets
8:     repeat until timeout
9:         $P, S \leftarrow Iterate(I, S)$                       ▷ Step (1)
10:        $P \leftarrow ModifyAndReplace(P, F, O, G)$
11:        if using angelic then
12:            $P \leftarrow AddAngelic(P, G, A)$
13:        if $P \in V$ then                                     ▷ Do not revisit programs
14:            continue
15:        $T \leftarrow GetPassedTests$(P, C)
16:        if $T = \emptyset$ then                               ▷ If passes at least one test
17:            continue
18:        if $P$ contains angelic conditions then               ▷ Step (2)
19:            $P \leftarrow ResolveAngelic$(P, F, G, T, C, A, O)
20:            if failed to resolve conditions then
21:                continue
22:            $T \leftarrow GetPassedTests$(P, C)
23:        $P \leftarrow Simplify$(P, G, T)                      ▷ Simplify program
24:        $T \leftarrow GetPassedTests$(P, G, C)
25:        if $T$ = $C$ then                                     ▷ Return program if all tests pass
26:            return $P$
27:        if $P$ is the simplest program to pass $T$ then       ▷ Step (3)
28:            $R \leftarrow RememberProgram$(P, R, T, F, G)
29:            $F \leftarrow MineFragments$(R)
30:            if $F$ is updated then
31:                $UpdateFragments(G, F, O)$
32:    return nothing
```

The pseudocode follows the main ideas and flow of the algorithm - the generation of a program, possibly with angelic conditions, which are placeholders for conditions for control flow structures in the grammar; evaluation of the generated program based on the provided spec; resolving any angelic conditions, and possibly mining fragments in case it is the simplest program so far that solves given subset of the spec. However, there are some differences:

Allows any context-free grammar. The generalised version can generate programs of any grammar, given a starting symbol, which allows the algorithm to be applied in different contexts and for various problems. However, the lack of default control structures, such as if statements, means the algorithm must also take as input a description of which nodes in which rules should be treated as possibly angelic. Additionally, expressing Java's complexity as context-free grammar is a challenge, which limits the algorithm usage in Java-specific problems that the original implementation can handle.

Furthermore, not using Java as the search space means that the algorithm implementation can no longer distinguish between statements, expressions, and other language constructs. This limitation affects the implementation of certain methods, such as advanced intermediate variable handling and context-specific simplification methods. While the $simplifyQuick$ method can be generalised, the $simplifySlow$ method is too context-specific and should be implemented based on the used context-free grammar.

Allows an iterator to be passed. The version accepts an iterator that can implement any traversal/generation strategy, allowing the program space to be explored in different ways compared to the original algorithm, which generates the new programs by using already mined fragments and choosing rules randomly. However, already existing iterators do not have the concept of fragments, and therefore, to be utilised, they are added to the context-free grammar itself, allowing iterators to choose them. Furthermore, the random percentages are also embedded in the grammar, enabling the iterator to decide whether to use the probabilities and adhere strictly to the entire algorithm. This approach provides flexibility and compatibility with various program search algorithms while maintaining the ability to utilise mined fragments effectively.

We propose the fragments to be added as two types of rules - a fragment placeholder, which connects an already existing symbol to a fragment of that symbol, and fragment rules, which contain the actual fragment expression. The split is necessary to allow making a distinction between a fragment and a grammar rule with the same expression as a value. An example of a fragment placeholder is $Num = Fragment\_Num$, while examples of fragment rules are $Fragment\_Num = 5 + x$ and $Fragment\_Num = 5$. Fragment placeholder rules are added for each symbol with an initial probability of zero. Once fragments for a symbol are added, the placeholder rule is assigned a probability equal to the chance of using a fragment.

To allow these changes, the generation of random programs is split into three subtasks:

(1) Generation of a program, which possibly includes fragment placeholders, with the provided iterator.

(2) Replacing all fragments placeholders and their inner fragments with the corresponding abstract syntax tree rule node, equivalent to the rule expression. During this swap, the fragments can be modified.

(3) The generated rule node tree is traversed, with the possibility of adding angelic conditions that replace concrete nodes with holes, indicating that any value of that symbol could be used.

Defining Program Synthesis from Rewards

The FrAngel algorithm takes input-output examples, but the MineRL environment only provides rewards. This raises the question of how to define program synthesis from rewards. To address this, we transform the reward problem into multiple output examples, acting as reward checkpoints that the output program reward should be equal to or greater. This enables the algorithm to mine fragments from programs reaching these checkpoints, which are in increasing order. For instance, in the navigation task, making progress in the right direction can be seen as a partially successful program, and useful actions can be then mined, which are connected to the solution. For example, if the expected reward is 64, we can set reward checkpoints at 16, 32, 48, and 64. If a program achieves a reward of 49, it will pass the first three checkpoints, resulting in the first three tests passing. A limitation of this approach is that the reward must be approximately known at the beginning to set the desired percentages of the maximum reward as checkpoints.

This definition also answers how to discover useful action in game environments. We identify useful actions based on the received rewards and, more precisely, by reaching a checkpoint. However, this solution relies on dense rewards and is not effective for sparse rewards, where rewards are only given upon task completion, negating the benefits of fragments and angelic conditions provided by the FrAngel algorithm.

Despite the algorithm's ability to find high-reward programs, finding the program that reaches the target location remains challenging and computationally expensive. To address this, we split the solution-finding process into multiple iterations of the algorithm, each building on the previous one. This approach allows the algorithm to search for a solution passing the output checkpoint or return a high-reward program for the task, defined as passing most output examples after a fixed amount of time. Then, the algorithm is run again, starting from the location that has been reached from the previous iteration. This approach allows the algorithm to focus on solutions for the remaining part without influence from prior fragments or actions, which allows, for example, turning into an entirely new direction from the last location to reach the goal without having a larger chance to pick a direction that used to lead to a higher reward until that moment. In each subsequent run, the required rewards in the spec tests are adjusted based on the achieved reward so far.

To balance the generation of high-reward programs with iteration until timeout, the required outputs for a solution are set to 80% of the expected remaining reward in the experiments. Other values have not been thoroughly tested, but this approach allows the algorithm to search without spending excessive time on complex programs that cannot be generated in one iteration. The approximated output reward should not be less than the solution reward to avoid considering negative-reward actions as useful.

Furthermore, to limit the possible search space, the grammar only contains actions related to moving, as seen in Algorithm 3. The mc_move function moves the player in the environment, given a direction, number of moves, and whether to sprint and jump. Additionally, the function only moves the player if the current total reward for the run is larger than -10 to stop programs that are most likely not leading to the solution. The mc_init softly resets the environment by teleporting the player to the given starting position for the iteration, while mc_end returns information about the last execution. There are also three boolean helper functions: $is\_done(state)$ returns if the goal has been reached, $mc\_has\_moved(state)$ verifies if the agent moved in the last action, and $mc\_was\_good\_move(state)$ which returns whether the last action had positive reward.

Algorithm 3 Grammar to represent possible MineRL programs for the navigation task. Each program starts with the Program symbol.

```
 1: Program = (state = Init; Blocks; End)
 2: Init = mc_init(start_pos)
 3: Blocks = Block | (Block; Blocks)
 4: Block = (while true
        Move ;
        Bool || break
    end)
 5: Block = Move
 6: Block = (if Bool Move end)
 7: Move = mc_move!(state, Direction, Times, Sprint,
    Jump)
 8: Direction = (["forward"]) | (["back"]) | (["left"]) |
    (["right"]) | (["forward", "left"]) | (["forward", "right"])
    | (["back", "left"]) | (["back", "right"])
 9: Sprint = true | false
10: Jump = true | false
11: Times = 1 | 2 | 3 | 4
12: Bool = is_done(state) | mc_was_good_move(state) |
    mc_has_moved(state) | !Bool
13: End = mc_end(state)
```

An example of fragments with the following grammar consists of anything from an entire program to a sequence of actions/moves in the game to a direction to move in. Although not all fragments will be useful at all times when generating new programs, the fragments can guide the rest of the search. For example, it is often the case that there are direction fragments only in the direction of the goal, as well as there are fragments for useful boolean conditions and small utility programs such as moving forwards until an obstacle is reached.

4 Experimental Setup and Results

This section explains how the experiments were run, shows the results and draws conclusions.

Setup

FrAngel implementation in Herb.jl. One contribution of this work, in collaboration with Alperen Guncan, a BSc student at TU Delft, is the generalised implementation of the FrAngel algorithm in Herb.jl, a Julia program synthesis library developed at TU Delft [12]. This implementation is then used as a base to conduct the experiments on.

The implementation follows the proposed changes to the original algorithm in Section 3, with the only further deviation that it introduces program caching to optimise its performance. Using caching for already visited programs improves traversal by excluding previously generated programs identified as non-solutions. For speed and memory efficiency, only the hashes of the programs are stored and compared.

When it comes to the used iterator, the implementation provides $FrAngelRandomIterator$ that simulates the exact generation process as described in the FrAngel paper.

MineRL [13] – Minecraft Environment. MineRL is a Python library integrating Minecraft with OpenAI Gym, offering agents tasks such as navigation, tree chopping, and resource gathering. This paper concentrates on the navigation task, where agents aim to reach a specified location from their starting point. The experiments use the version of the task with dense rewards, where the agent receives a reward proportional to the distance moved toward or away from the goal. Although the environment provides visual data, utilising it requires image processing and is therefore out of the scope of this paper.

Experiment configuration. Due to Minecraft's procedural generation, five world seeds, with different features, are chosen initially to run the experiments on. Furthermore, due to FrAngel relying on random number generation, a set of 8 predefined random generator seeds is chosen before any of the experiments are run to avoid bias.

For the experiments, each run is limited to 300 seconds, with each algorithm iteration limited to 40 seconds. Output checkpoints are set at each 10% increment up to 80% of the expected remaining reward, which is set to 74 based on one of the worlds. The maximum program size is limited to 40 AST nodes, with a 0.5 probability of using fragments and a 0.2 probability for angelic conditions. The boolean expression max size is set to 6, and angelic replace attempts are limited to 4 due to the limited number of boolean options in the grammar. Furthermore, the FrAngel implementation is modified to provide more logging data about each run and to support the concept of the best-found program so far, the one passing most tests, which is required in this definition of the problem.

Running the experiments. Instructions for setting up the environment can be found on GitHub: Julia and Herb.jl installation instructions[1] and MineRL setup instructions[2]. The code and run instructions are also available on GitHub here[3].

The specifications of the computer that ran the experiments in this paper are the following:

[1]Julia and Herb.jl installation instructions can be found here
[2]Instructions on how to set up MineRL.
[3]Code and run instructions here.

- Processor: AMD Ryzen 7 4800H
- RAM: 32 GB of DDR4
- GPU: NVIDIA GeForce GTX 1650
- Operating system: Windows 11
- Julia version: 1.10.3+0.x64.w64.mingw32

Running all of the experiments takes approximately 28 hours on the mentioned machine due to the many combinations of world and random seeds for each experiment. Consequently, because of computational limitations and the variety of random seeds and Minecraft worlds, the results are limited and may not fully represent the overall performance of the changes.

Experiment 1 – Different Beginning Reward Checkpoints

Experiment setting One of the crucial factors for the effectiveness of the FrAngel algorithm is the provided input-output examples and the fragments that they lead to. The fragments that are then to be exploited must be relevant to the solution, and from this comes the question of how much exploration is needed before considering a fragment as useful.

In this experiment, the smallest reward checkpoints are increased and decreased, such that the length of the pure exploration part of the FrAngel algorithm could be increased or decreased. This should eventually provide different quality fragments that can be explored after that, possibly influencing the entire run.

For this experiment, four additional setups are compared to the base setup. One reduces the first reward checkpoint to 5% of the expected reward, compared to the default of 10%. The other three setups have their lowest checkpoints set at 20%, 30%, and 40% of the expected reward.

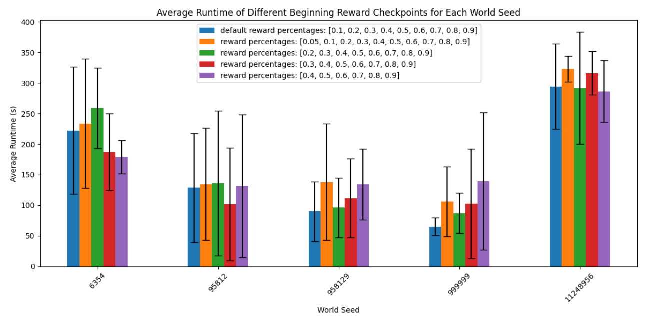


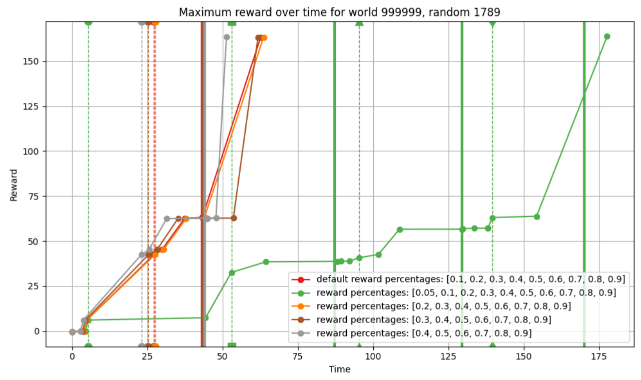
Figure 3: Average runtime of different beginning reward checkpoints for each world seed.

Interpretation of results From Figure 3, it can be seen that the runtime depends highly on the particular world seed and that different reward checkpoints have different effectiveness based on the seed combination. However, it can also be seen that, on average, the default case with rewards starting at 10% performs the best on average across all worlds.

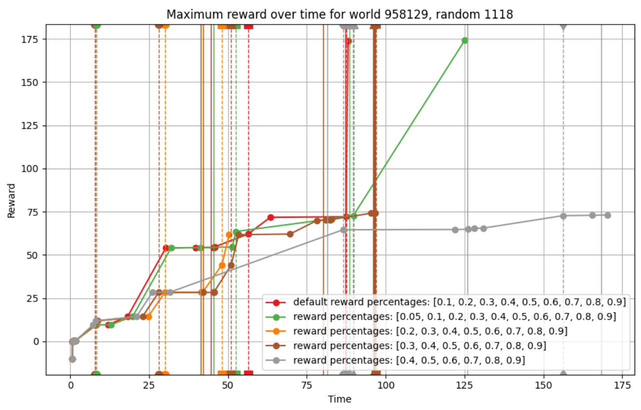
Selecting the appropriate threshold value to determine the usefulness of a program is crucial as it guides the rest of the search within that checkpoint. Finding useful components and using them marks the end of the full exploration part of the FrAngel algorithm with the combination of exploration and exploitation. Therefore, it is essential to avoid extracting fragments that cannot be effectively utilised, as they do not contribute significantly to the solution, as illustrated in Figure 4a. In this scenario depicted by the green line, FrAngel attempts to exploit previously generated programs that are not sufficiently useful, resulting in long search times until a valuable solution is discovered. This highlights one of the potential drawbacks of using fragments in the algorithm—they are only useful as the output examples are. Badly chosen fragments can slow down the search process.

There is also the other extreme, where only really useful programs are considered to be useful, which could lead to a long period of exploration until such a program is found, and therefore only relying more on the initial program generation. Visualized by the grey line in Figure 4b, which continues the search.

This experiment shows the importance of choosing a lower bound/easy output example that is both not too easy and too hard.



(a) world seed - 999999, random seed - 6354



(b) world seed - 958129, random seed - 1118

Figure 4: Maximum rewards over time for different seed combinations. Full lines are checkpoints, dotted lines indicate first fragment use.

Experiment 2 – Limited Fragment Symbols

Experiment setting Another question related to fragments and exploration is whether limiting fragments to only some of the lower grammar symbols could potentially speed up the program synthesis by forcing the exploration of diverse sequences of actions, both useful and random.

In this experiment, symbols defining entire programs and sequences of actions are not mined. However, the remaining mined symbols still allow the discovery of useful blocks like moving in a certain direction or moving in a loop. This approach aims to promote the exploration of various potential program sequences rather than repeating more basic ones that are already a fragment.

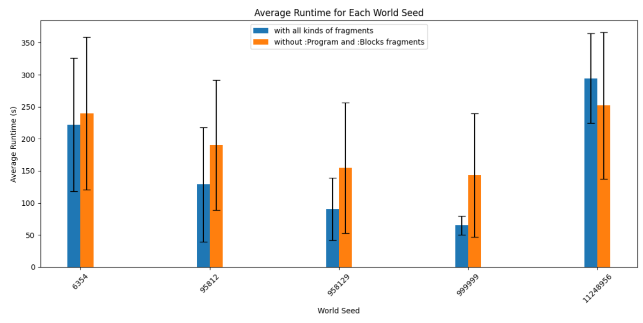


Figure 5: Average runtime with all and with limited symbols as fragments for each world seed.

Interpretation of results Again, in Figure 5, we can see that the runtime and impact of the change depend on the combination of world and random seed. However, mining all symbols shows better performance overall. I assume this is the case due to the nature of the task – navigation, where many actions/fragments are only useful from a particular starting point, and therefore, generating a different sequence of them makes them less applicable. This experiment shows that fragments are often context-dependent, and therefore, by removing larger fragments that are closer to representing an entire program, we are no longer exploiting what was already useful but rather trying to explore something that uses it, which is not necessarily an easy or efficient task.

Experiment 3 – Initially Generating More Diverse Programs

Experiment setting By generating programs randomly, it is possible to use certain rules more often than others, and from that arises the question of whether this could negatively impact the exploration and, consequently, the entire program synthesis process.

In this experiment, the random iterator is modified to give a higher probability to basic rules used in generation less frequently. This aims to determine if this approach accelerates the exploration of more useful fragments/rules. The probabilities of the possible rules are multiplied by $1/(1+n)$ and then normalised, where n is the number of times the rules have been used in program generation so far. These changes in probabilities will only affect the non-fragment rules, while the chance to use a fragment will remain as configured.

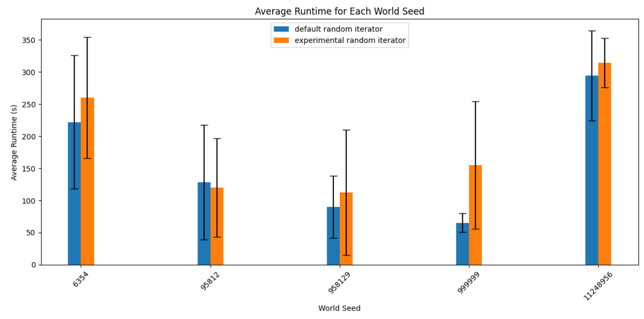


Figure 6: Average runtime with the default random iterator and the modified one for each world seed.

Interpretation of results As shown in Figure 6, in most of the runs the default random iterator performs better. It could be possibly due to the low amount of iterations of each checkpoint FrAngel run, such that the change of iterator does not take as much difference, the random seeds, or it is simply not more effective.

Experiment 4 – Fragments Usage and Modification Probability

Experiment setting Another question that stands is whether increasing exploration in the second phase of the FrAngel algorithm, which involves both exploration and exploitation, improves the algorithm's performance.

In this experiment, six different setups are tested with varying probabilities for fragment modification and usage. For fragment usage, two values are chosen: 0.3 and 0.5. For fragment modification chance, three values are used: 0.25, 0.5, and 0.75.

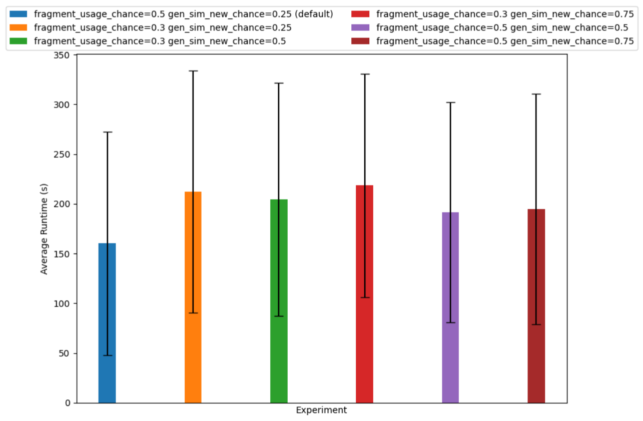


Figure 7: Average runtime of all runs for each fragment configuration. fragment_usage_change is the chance to generate the current node with a fragment, while gen_sim_new_chance is the chance to modify a node in a fragment.

Interpretation of results From the conducted experiment runs, highlighted in Figure 7, we can conclude that the default amount of usage - 0.5 and the default amount of modification - 0.25, as specified in the FrAngel paper, performs best on average. While other configurations may perform more exploration by reducing the fragment usage or increasing the modifications, they also decrease the exploitation, which increases the time it takes to solve the task.

5 Responsible Research

This section addresses concerns regarding the reproducibility of the experiments and the credibility of the results and findings.

The reproducibility of the experiments depends on two main factors: the experimental setup and the processing power of the machine running them. All code, along with instructions on how to run the experiments, can be found on GitHub[4]. Also, the repository includes all results from the runs and the corresponding seeds. Furthermore, to make it more accessible, the experiments were conducted on a personal computer, with specifications detailed in Section 4. Using a fixed set of seeds for random number generation ensures consistent replication of the results, although minor variations in runtime may occur due to differences in computational power. To further ensure reproducibility, the experiment results include the exact number of algorithm iterations in each case.

Due to computational limitations, the experiments have only been run on a small set of environments and seeds for random number generation. Although the seeds were chosen before conducting any experiments to avoid cherry-picking and bias, it is still possible that the chosen seeds do not fully represent the general case, potentially influencing the results. Therefore, future research needs to extend the experiments to a broader range of environments.

6 Conclusions and Future Work

This paper presented a generalised implementation of the FrAngel program synthesis algorithm in Herb.jl, allowing any context-free grammar and iterator. Additionally, it defined program synthesis from rewards by modelling reward checkpoints as input-output examples and outlined a general algorithm for applying the FrAngel algorithm to the navigation task in Minecraft, where a location must be reached based on a dense reward.

Various modifications to FrAngel were compared in the context of exploration. It was found that it is crucial to balance the difficulty of the simpler specification tests. If these tests are too easy, the found fragments may not be useful enough and could guide the search in the wrong direction. On the other hand, if the tests are too difficult, it becomes harder to find promising programs to exploit. Another algorithm modification involved limiting the symbols from which fragments can be mined. Although this change increased exploration, it restricted the use of context-specific fragments, rendering them less useful when they were no longer part of the larger program.

However, the results from these comparisons are not definitive due to the experiments being conducted on a limited set of environments and random number generation seeds. Future work should expand the range of environments to allow for more general conclusions. Additionally, the algorithm should be applied to other games and tasks and possibly to problems with sparser rewards to test its generalisability and potentially discover additional findings.

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

References

[1] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017.

[2] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, Oct. 2019.

[3] Q.-Y. Yin, J. Yang, K.-Q. Huang, M.-J. Zhao, W.-C. Ni, B. Liang, Y. Huang, S. Wu, and L. Wang, "AI in human-computer gaming: Techniques, challenges and opportunities," *Machine Intelligence Research*, vol. 20, no. 3, pp. 299–317, Jun. 2023.

[4] C. Hu, Y. Zhao, Z. Wang, H. Du, and J. Liu, "Game-based platforms for artificial intelligence research," 2024.

[5] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Foundations and Trends in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017. [Online]. Available: http://dx.doi.org/10.1561/2500000010

[6] S. Jacindha, G. Abishek, and P. Vasuki, "Program synthesis—a survey," in *Computational Intelligence in Machine Learning*, A. Kumar, J. M. Zurada, V. K. Gunjan, and R. Balasubramanian, Eds. Singapore: Springer Nature Singapore, 2022, pp. 409–421.

[7] K. Ferdowsifard, S. Barke, H. Peleg, S. Lerner, and N. Polikarpova, "Loopy: interactive program synthesis with control structures," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485530

[8] 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

[9] N. Natarajan, A. Karthikeyan, P. Jain, I. Radicek, S. Rajamani, S. Gulwani, and J. Gehrke, "Programming by rewards," 2020.

[10] P. Ladosz, L. Weng, M. Kim, and H. Oh, "Exploration in deep reinforcement learning: A survey," *Information Fusion*, vol. 85, pp. 1–22, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1566253522000288

[11] 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

[12] 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

[13] W. H. Guss, B. Houghton, N. Topin, P. Wang, C. Codel, M. Veloso, and R. Salakhutdinov, "Minerl: A large-scale dataset of minecraft demonstrations," 2019.