Program Synthesis from Game Rewards Using FrAngel

Finding Complex Subprograms for Solving Minecraft

Alperen Guncan

Supervisors: Sebastijan Dumančić, Tilman Hinnerichs

EEMCS, Delft University of Technology, The Netherlands

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

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

Abstract

Program synthesis has been extensively used for automating code-related tasks, but it has yet to be applied in the realm of reward-based games. FrAngel is a component-based program synthesizer that addresses the aspects of exploration and exploitation, both important for the performance of a program synthesizer. However, its specific implementation makes it hard to study and extend. We first implement a generalized version of FrAngel that takes arbitrary grammars and program generators, while also generalizing most of its features by redesign. We then use it to introduce a novel approach to formulating a reward-based problem into an inductive specification. We integrate this algorithm within MineRL, an AI framework for playing Minecraft. Lastly, we lay the groundwork for using program synthesis for Minecraft by investigating the ways of tuning the algorithm to generate complex subprograms. By changing the configuration parameters, as well as slightly changing the algorithm's implementation, we managed to observe a significant increase in the complexity of fragments and generated programs. These modifications are a stepping stone towards using program synthesis to solve complex game tasks.

1 Introduction

The task of automatically generating programs based on user requirements has always been of high interest to researchers and software engineers alike. One of the more researched methodologies is through program synthesis. This is where a program automatically finds other programs that satisfy a user specification. In its essence, program synthesis searches over the space of all possible programs, usually called the *program space* [1]. For each candidate in the space, it checks if it passes the user's specification. The specification can take many forms, most commonly a list of examples outlining the program's expected behavior [1].

While it has its applications in data processing and graphics, program synthesis is mostly used for coding-related tasks, like code repair, super-optimization, and automatic code generation [1, pp. 15-34] [2]. The last application is especially sought after. The hope is that the process of writing formally correct subroutines is fully automated. With current advancements, there are already program synthesizers that can generate programs for subsets of generic programming languages, like C [3].

FrAngel is another program synthesizer that strives to generate generic programs, with a multitude of features [4]. FrAngel can reuse subprograms in future stages and prune its search in smart ways. Its goal is to allow for the generation of complex programs with arbitrary control statements and library functions [4]. However, the algorithm relies on specific implementation details. They essentially hinder its ability to be generalized. This makes it difficult to extend, adapt to specific problems, and study as a whole. This issue is common in the field of program synthesis - the lack of standardized tooling for implementing and benchmarking synthesis problems.

The paper firstly discusses how we generically reimplement FrAngel with an existing library, namely 'Herb.jl' [5]. This library aims to resolve the issue of non-standardization in the field. It does so by generalizing some aspects of implementing a program synthesizer, while also standardizing others. We go over the changes made during generalization and also mention additions to the algorithm.

We then apply this algorithm in the domain of playing reward-based games. This type of research serves more than philosophical curiosity. Research on artificial intelligence in games drives advancements in optimization, reinforcement learning, and decision-making. This lies in its ability to push the boundaries of what is achievable by AI agents [6]. Prior research on program synthesis for games is limited, mostly focusing on zero-sum games [7]. However, this excludes many other game types, including most single-player and cooperative games. This paper explores a broader set of reward-based games by applying the FrAngel synthesizer to play Minecraft, an all-time best-seller sandbox game that offers a diverse range of tasks [8].

The paper explores a novel problem formulation for playing games with program synthesis. The program specification for FrAngel needs to be remodeled to account for reward-based specifications. They differ significantly from the usual input-output examples by their properties. Furthermore, to fully utilize FrAngel's capabilities, we need to define what constitutes a 'useful' program and how to find them. Ultimately, we apply our findings to generate complex subprograms for solving Minecraft tasks. The research examines how to tune FrAngel to achieve the goal of complex program generation. Due to Minecraft's rich environment, a program synthesizer must find complex solutions to properly capture the task complexities.

To test the implementation, we integrate our synthesizer with MineRL, the largest and most supported AI framework for playing Minecraft [9]. We benchmark the base and tuned models on MineRL and compare the complexity of generated programs and fragments across multiple configurations.

2 Background

A program synthesis problem is generally defined by two elements: the specification that it needs to fulfill and the program space to search over [1]. In its inception, the specifications were formal and highly mathematical. Automatic algorithm generation dates back to the 1930s, with Kolmogorov's work on interpretable subprograms [10]. There, programs are generated from formal proofs of the user's specifications. This approach of specification is called *deductive*. This approach extracts the logic from formally correct proofs to generate useful programs. Although initially it worked well, it ended up having a major issue. For more complex programs, writing the formal specification is just as difficult as writing the algorithm directly [1]. It takes high mathematical prowess from the user to successfully write a correct formal specification [2].

Inductive program synthesis

To address the issue of difficult specification, recent research led to a new method of specifying behavior - an *inductive* approach. This is the process of generating programs based on a limited specification. The synthesizer *induces* the generally desired behavior, hence its name. The partial specification is defined by *input-output example pairs* [1]. Unlike deductive approaches, I/O examples are easy to use and flexible, making it an effective approach to formalize program behavior. The inductive approach dates back to the 1970s

when it was used to generate small LISP programs [11]. However, it has only recently gained recognition among researchers [1]. One famous example is its use for the FlashFill algorithm - the paper emphasizes its intuitive and simple design, along with its direct, tangible impact on the synthesizer [12].

Syntax-guided synthesis

In inductive program synthesis, the structure of programs is given by a *grammar*. Only recently Rajeev et al. [13] introduced the use of syntactic templates to constrain the search, directly provided by the user. It specifies the allowed syntax and how programs are generated. By design, the programs we generate are always syntactically correct. The grammars can either be context-free, context-sensitive, or even probabilistic. In the latter case, we specify the probability of picking each rule for generation.

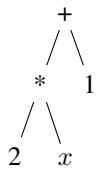
Now that we have introduced the main elements of program synthesis, we give a small example problem and a solution. Let us say that we want to generate a linear function as an expression. In Figure 1 we show the formulation of this problem.

$(x = 1) \rightarrow 3$
$(x = 2) \rightarrow 5$
$(x = 3) \rightarrow 7$

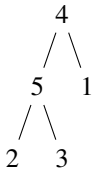
(a) Input/Output specification

$1/2.\ Num = 1|2$
$3.\ Num = x$
$4.\ Num = Num + Num$
$5.\ Num = Num * Num$

(b) Grammar for linear functions



(c) AST for $2x + 1$



(d) AST for $2x + 1$ as `RuleNodes`

Figure 1: A basic Program Synthesis Example

In (a), we specify the behavior inductively, by listing the expected output for several inputs. In (b), we give the syntactic specification, or grammar, that can be used to generate the program. In (c) we show a solution to our I/O specification as an abstract syntax tree. More generally, we represent the tree with the grammar rule indices, as in (d). For instance, note that the 4th rule is the addition of two numbers - the topmost expression in (c). In this way, we can represent any solution in a grammar-agnostic way. We call the nodes of this tree *rule nodes*.

Enumerative search

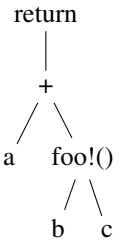
Another variable in implementing a program synthesizer is its search technique. This is how the algorithm looks for candidates within the program space. Most commonly, this is done through an *enumerative search*. This is a technique that *enumerates* the program space in a certain fashion, and checks the specification for each candidate. A naive search would go over all possible programs. This is an intractable problem, as candidates grow exponentially with their tree size [1] [2]. The performance of a program synthesizer is often summarized by how smartly it prunes its search, and the quality of the candidates it finds.

The efficiency of a synthesizer is gauged by the balance between its ability to *explore* and *exploit* its program space. Exploration is its ability to find new good candidates in the unexplored program space. Exploitation, on the other hand, is its ability to augment new programs by reusing parts of previous ones. A decent program synthesizer should have features that address each of these challenges.

FrAngel

FrAngel is a modern *component-based* program synthesizer that aims to generate highly complex and generic algorithms [4]. A component-based program synthesizer is any synthesizer that uses a set of libraries provided as its building blocks for generating programs. FrAngel offers two key innovations to address the challenges of exploitation and exploration: fragments and angelic conditions.

Fragments are FrAngel's way of reusing useful subprograms. In each iteration, the generated program is checked for whether it passes at least a single I/O pair. If it does, it is deemed "useful", as it contains at least one sub-expression that caused it to pass said test[s]. All such programs are *remembered*.



(a) AST

```
a
b
c
foo!(b, c)
a + foo!(b, c)
return a + foo!(b, c)
```

(b) Mined fragments

Figure 2: An example of the *fragment mining* process.

On each iteration, we get all their subprograms in a process called *fragment mining*. This process is showcased in Figure 2 for the example program $return\ a + b.foo(c)$. The algorithm may modify the structure of each generated program by replacing some subtrees with these fragments. However, FrAngel does not store every program that passes a single test. If two programs pass the same subset of tests, they most likely share a similar [sub-] structure. Thus, to maximize relevance, only the simplest program is *remembered*.

Angelic conditions are FrAngel's answer to exploration. These are placeholder expressions that generalize control statements to be resolved at the end. This allows FrAngel to search over the program space more quickly. This, in turn, helps it find useful expressions more often. Control statements (if-statements and loops) have the strongest effect on a program's behavior. There is a big difference between, for example, a loop being skipped, going for 20 iterations, or running forever. As such, the algorithm does not try to resolve these statements on the first attempt. Instead, it puts

placeholders on these locations. This is shown as an example in Figure 3 [4, Fig. 6]. During evaluation, the same program is checked multiple times, by only changing the control statements.

```
double sumPositiveDoubles(double[] arr) {
    double sum = 0.0;
    for (int i = 0; <ANGELIC>; i++)
        if (<ANGELIC>)
            sum = sum + arr[i];
    return sum;
}
```

Figure 3: An example angelic program with two conditions.

With both of these features, FrAngel is a powerful program synthesizer that can generate multiline complex programs, with loops, conditional statements, and library functions. We use this sophisticated synthesizer to tackle the domain of playing reward-based games.

3 Methodology

We first describe the issues of the existing implementation of FrAngel. Then, we address and justify our decisions for generalizing the algorithm. Afterward, we modify the FrAngel program synthesizer to be used for reward-based games. The adaptation process has two main challenges - how we test the generated program, and how we explore the game environment for useful programs. After proposing a solution to each, we delve into the final research question and describe the experiments to be run.

Generalizing FrAngel

The original implementation of the algorithm, given in the FrAngel repository [14] is written in Java, with many specific design choices. They make the algorithm difficult to customize and study. The implementation has such issues in most aspects of a program synthesizer - from its design of the grammar to the generation of programs. We will go over the three main conceptual subroutines and how we generalize each: program generation, fragment mining/storage, and angelic conditions. A pseudo-code to summarize the generalization on a high level is shown in Algorithm 1. Additions and changes to the original algorithm have been highlighted for simplicity. The rest is made to resemble the original as closely as possible.

Probably the most significant issue is in the way the grammar and programs are defined. In the original implementation, both of these are hard-coded into the solver. Expressions and statements are each implemented as separate objects, combined with complex logic. This means it is very difficult to extend the grammar and program structures. The solver provides only the minimal configuration needed, namely the input/output parameters and allowed library functions. The algorithm cannot take any other grammar than what is hard-coded; this heavily restricts its use for problems better modeled with domain-specific languages (like the one for Minecraft). We will let the user provide the grammar and program generator (also called *iterator* in 'Herb.jl') instead. This flexibility enables solving problems

Algorithm 1: Outline for generalized FrAngel - Herb.jl

Input: *Grammar G, program iterator I, set of test cases C, angelic conditions A*
Output: *The best-performing program P that passes the most tests in C, or Nothing if no program was found to pass a single one*

procedure FrAngel(G, I, C, A)
  $R \leftarrow \emptyset$ ▷ A set of remembered programs
  $F \leftarrow \emptyset$ ▷ A set of fragments
  $V \leftarrow \emptyset$ ▷ A set of visited programs
  $S \leftarrow Nothing$ ▷ Initialize state for iterator
  $B \leftarrow Nothing$ ▷ Best partial solution

  $AddAngelicRuleNode(G)$
  $O \leftarrow AddFragments(G)$ ▷ Store fragment rule offsets
  repeat until timeout
    $P, S \leftarrow Iterate(I, S)$ ▷ Generate & modify program
    $P \leftarrow ModifyAndReplace(P, F, O, G)$
    if using angelic this iteration **then**
      $P \leftarrow AddAngelic(P, G, A)$
    if $P \in V$ **then** ▷ Do not revisit a program
      continue
    $T \leftarrow GetPassedTests(P, G, C)$
    if $T = \emptyset$ **then** ▷ Check if it passes at least one test
      continue
    if P has an angelic condition **then** ▷ Angelic
      $P \leftarrow ResolveAngelic(P, F, G, T, C, A, O)$
      if P still has an angelic condition **then**
        continue
    $P \leftarrow Simplify(P, G, T)$ ▷ Simplify program
    $T \leftarrow GetPassedTests(P, G, C)$
    if $T = C$ **then** ▷ Return if passes all tests
      return P
    else
      $B \leftarrow UpdateBestProgram(B, P, T)$
    if P is simplest to pass T **then** ▷ Fragments
      $R \leftarrow RememberPrograms(R, P, T, F, G)$
      $F \leftarrow MineFragments(R)$
      if F is updated **then**
        $UpdateFragmentsInGrammar(G, F, O)$
  return B

with either generic or specific/domain-based grammar and generators, all up to the user's needs.

We then focus on the two main features of FrAngel - fragments and angelic conditions. The obvious approach to implementing fragments is to strictly follow the FrAngel specification - store all the fragments, and only consider using them during program modification. However, it would improve the performance, and provide more variety in generated programs, if they can be used by the program iterator directly. With arbitrary grammars and iterators, it is non-trivial to incorporate them into program generation. One option is to re-implement all existing iterators to optionally accept fragments. This is too costly in terms of development and would over-complicate the design of program iterators. Instead, we will add fragments directly into the grammar. With this, all iterators can optionally use the fragments as rules in the grammar. Most importantly, this enables more diversity in generated programs.

The angelic conditions are also implemented quite forcibly. In the original FrAngel, control statements have a toggle field to determine if they are angelic. This bloats them

with data that is redundant in most cases (angelic conditions seldom occur by the original spec). Furthermore, only some statements have that toggle, so the locations are limited. We can implement them by changing how programs are defined in the core module. Yet, to avoid making changes to the entire library, we decided to represent angelic conditions differently. Similarly to fragments, we will have a rule in the grammar. This will serve as a placeholder for angelic conditions. Most importantly, this allows for putting angelic conditions practically everywhere among the syntax tree of a program. This leads to very flexible program evaluation and generation. It also minimizes the changes that need to be made for interpreting programs.

Extra additions/changes needed to be considered as well. Currently, 'Herb.jl' does not have the possibility of returning code paths during interpretation - a required functionality for angelic conditions. Furthermore, to play games more efficiently, we will allow FrAngel to return partial solutions. Finally, we have an important consideration regarding program generation. Once we make the grammar user-defined, it can range from very simple grammar with only a few rules, to an extensive, or even Turing-complete language. In the former's case, program "collisions" are very likely. It means that a program may be generated and analyzed multiple times. Yet, if a program has been visited once, future re-analysis cannot improve the state of the search. Our implementation will keep track of visited programs during their execution to avoid this redundant work.

Defining specification by reward

The main challenge in solving reward-based game tasks with program synthesis is formulating the problem correctly. With an inductive specification, it is unclear how the I/O examples will be defined. A naive inductive specification would not work on its own - to justify this, consider the uniqueness of program solutions in either case.

In inductive program synthesis, the output remains the same, specifically what the user specified with an example. In contrast, reward-based game tasks can have numerous solutions to the same task. A basic example is navigation - the goal is static, while the path to reach it is not. In short, there are potentially infinite programs, each generating its path and altering the environment differently, but still solving the task. Furthermore, to check the generated programs, we have to run them in the Minecraft environment and alter the game state. This requires a different approach to testing, as running and checking programs becomes a performance bottleneck. We aim to minimize repeated evaluations. It is also why we avoid angelic conditions in most experiments.

The method we use for creating a spec is by splitting the maximal reward into segments. In turn, each one comprises a separate test case. Then, the program is run in the Minecraft environment, and the final reward is compared across all tests. A program passes a test if its final reward is greater than or equal to the test case's specified reward. As such, if the player completes the task, and acquires the maximal reward, the program will pass all tests.

$$max\_reward = 50$$
$$percentage\_splits = [0.25, 0.5, 0.75, 1]$$
$$spec \leftarrow CreateSpec(max\_reward, percentage\_splits)$$

As an example, the snippet above will result in a vector of 4 IOExamples. Each has the starting player's position as input and for outputs - 12.5, 25, 37.5, and 50 respectively.

Exploring the game environment

It is also important to define how we are going to use the programs that FrAngel generates. By extension, we need to figure out how the environment will guide its search. MineRL's main source of information is through imaging/screen information of the player. However, image processing techniques are complex and out-of-scope, thus we only make use of the numerical reward.

Given how complex MineRL tasks can be, it makes sense to split the whole task into subtasks. The most natural way of doing this is by "checkpointing". Here, this means we periodically store the state with the highest reward yet. With that, we can force the player to start his search from there. To incorporate FrAngel into MineRL, we will generate programs repeatedly while keeping track of the best-found state so far. At the end, we make a checkpoint and start the next iteration from the new position/state.

Generating complex subprograms for Minecraft

Finally, we will look into how FrAngel can be tuned to generate complex subprograms to play Minecraft. In our experiments, we consider one subprogram *more complex* than another if its syntax tree is larger by the number of nodes. This encapsulates complexity in the number of subexpressions. By extension, it also accounts for the individual sub-expression complexities. We will keep track of *two main metrics* - the complexity of the generated programs, and all fragments. Our goal is to tune FrAngel to generate more complex programs and fragments, while not wrecking performance.

We split the experiments into two types - ones that only tweak the configuration of FrAngel and ones that change its original specification or function. For the former, we focus on the fragments, since complex subprograms tie heavily with exploitation. We will consider two types of config changes, based on the "quantity vs. quality" paradigm. Respectively, they refer to changes in how large programs are, and how often they are augmented with fragments. Then, we turn our attention to the "fragment mining" feature. Originally, FrAngel stores the simpler programs to resolve ties between candidates. We look into flipping this condition, with the hope that storing the more complex programs will offer a wider range of fragments and more complex programs. Finally, we consider the idea of changing the rule preference of the algorithm to choose recursive grammar rules, like

$$Statement = (Statement; Statement)$$

more often. This forcibly makes FrAngel explore larger programs.

4 Experimental Setup and Results

We first describe how we implement FrAngel within the 'Herb.jl' library. Then, we introduce the experimental setup for the last research question. Finally, for each experiment, we describe our hypothesis and discuss the results.

Implementing generalized FrAngel

After the iterator has generated a program, we modify it by following the original FrAngel specification: we modify subtrees, look for replacements with new programs and fragments, and add angelic conditions wherever possible. This behavior can be configured with a 'config' struct that is also passed to the algorithm.

Once we add fragments to the grammar, we calculate the probabilities for the grammar. We let the algorithm pick fragments uniformly across the same type. To easily update the grammar, we have two types of 'fragment rules': the *identity-fragment* rules, and the *regular fragment* rules. The former interconnects fragments with their base type so that they can be used interchangeably. An example for the type *:Num* would be

$$Num = Fragment\_Num$$

The regular fragment rules map each fragment type to all its fragment expressions and adjust their probabilities of selection. For example, if we had the fragment "$5 + x$", and its return type is *:Num*, we would have the rule

$$Fragment\_Num = :(5 + x)$$

We provide a simple example of how the grammar would look after an arbitrary iteration, in Figure 4. Right before the first iteration, the grammar adds the identity fragment rules. Now, let the set of remembered programs after an iteration be $[(5 + x), (x == 3)]$. We do not only add the full expressions onto the grammar but also their complete subprograms. We *mine* fragments by taking the root expression, and all its subsequent complete subprograms, while also accounting for duplicates. In the above example, the variable `:x` is used in both expressions, but only added once to the grammar.

Base grammar
```
Num = |(0:10)
Num = (Num + Num) | (Num - Num) | x
Bool = (Num == Num) | (Num < Num)
```
Identity rules
```
Num = Fragment_Num
Bool = Fragment_Bool
```
Regular rules
```
Fragment_Num = (5 + x) | 5 | x | 3
Fragment_Bool = (x == 3)
```

Figure 4: An example grammar during an arbitrary FrAngel iteration, modified with fragments (identity & regular rules).

For the angelic rule node placeholder, the algorithm appends the rule to the grammar automatically. Our implementation defaults to

$$Angelic = update\_angelic\_path$$

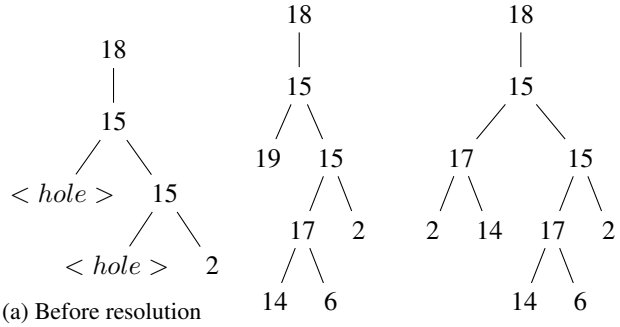
We ensure that the rule has zero probability, as it does not have any true meaning outside of angelic execution. We provide an example of how angelic resolution and execution work with an example. We assume the grammar in Figure 5, similar to the one used for describing the fragments above, extended with *if-statements*:

Let us say that the iterator generates a `Program` with two if-statements, one nested within the other, and a return value of 1. Assume both conditionals are replaced with angelic

```
Num = |(0:10)
Num = (Num + Num) | (Num - Num) | x
Num = (if Bool ; Num ; end)
Bool = (Num == Num) | (Num < Num)
Program = Num
Angelic = update_angelic_path
```

Figure 5: The grammar for our angelic example, with the angelic rule node and conditional statements

placeholders. This program is represented as a syntax tree in Figure 6. We also show all the intermediate steps of resolution and execution.



(a) Before resolution (b) First resolution (c) Fully resolved

Figure 6: An angelic expression's state in our FrAngel

Initially, we add `AbstractHole` nodes where the angelic condition would reside (a). During resolution, we attempt to replace each hole with a `Boolean` expression, one at a time (b). Right before an angelic evaluation, we replace all the remaining, unresolved holes with the angelic rule node, which is `RuleNode(19)` for this grammar. We repeat this process until we find replacements for all holes (c). The distinction between holes and angelic rule nodes is important, both for correctness and performance reasons. This is because the domains of holes change every iteration in which the grammar is updated with fragments. This indirectly entails that equivalent programs with holes will have different hashes, and will thus be revisited. Importantly, this is the conceptually correct approach - non-angelic programs offer no benefit when revisited, but angelic programs do. The `Bool` replacements are randomly generated and thus may be different on each visit.

Game environment and MineRL integration

The general algorithm that we use in MineRL to run program synthesis is shown in 2.

This subroutine is run for every world seed, and FrAngel seed (for randomization), for each experiment run separately. The world seed is what Minecraft uses to generate the in-game environment. Note that the I/O examples are generated anew on every iteration. The closer the task is to completion, the more of the initial tests are meaningless. This is because a good checkpoint will already pass the first few reward checkpoints. Thus, the specification ranges from the current reward, instead of 0, to the goal's reward.

For our experiments, the environment we focus on is

Algorithm 2: Running MineRL experiments with FrAngel

Input: World seed W, FrAngel seed R, MineRL config S
Output: A program P that achieves the task (equivalent to passing the final test case in C)

procedure FRANGELWITHMINERL(W, R, S)
  $G, A \leftarrow GetMinecraftGrammar()$
  $I \leftarrow FrAngelIterator(R)$
  repeat until timeout
    $C \leftarrow GenerateSpec(S)$
    $P \leftarrow FrAngel(G, I, C, A)$
    if $P \neq Nothing$ **then**
      if task is completed then
        break
      $UpdateStartingPositionAndReward()$
  $SaveData()$

`MineRLNavigateDenseProgSynth`. It is a slightly modified version of the original dense navigation task in MineRL. The goal is to find a diamond block that is maximally 64 blocks away from the player. After each frame, a reward is calculated. It is inversely proportional to the distance of the player to the goal. This is the reward used for generating the specification.

The grammar we use for the experiments is listed below in Figure 7. We use a helper function `mc_move!` to generalize the player's actions. More importantly, the grammar includes control statements to fully utilize FrAngel's feature set. Loops are especially useful. The player usually must perform a pattern of actions repeatedly, like moving forward. Fragments can capture this behavior (by extracting the loop body as a subprogram). They can also store the conditional expressions if useful, for example, moving until the player is blocked (mc_has_moved).

```
Program = ( state = mc_init(start_pos) ;
     Stmt ; mc_end(state) )
Stmt = mc_move!(state, Dir, Times,
    Sprint, Jump)
Stmt = (Stmt ; Stmt)
Stmt = (if Bool ; Stmt ; end)
Stmt = (while Bool ; Stmt ; end)
Dir = (["forward"]) | (["back"]) | (["
    left"]) | (["right"]) | (["forward",
    "left"]) | (["forward", "right"]) |
    (["back", "left"]) | (["back", "right
    "])
Sprint = false | true
Jump = false | true
Times = 1 | 2 | 3 | 4
Bool = is_done(state) | mc_has_moved(
    state) | mc_was_good_move(state) | !
    Bool
```

Figure 7: The grammar used for the navigation task experiments with MineRL

The hardware for running the experiments has an `Intel Core i7-8700k`, and `32GB` of `RAM`. For the FrAngel seed, we stick to the same value for all tests to preserve reproducibility - 1234. Finally, we describe the five Minecraft world seeds used and a summarized description of what they pose as a challenge to the player below:

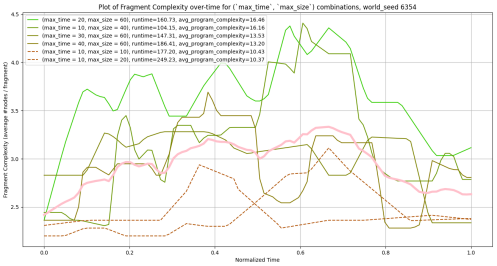
6354 → "Many trees. Small hill. Ocean on the way and goal on island",
958129 → "Relatively flat. Some trees. Small cave opening.",
95812 → "Big hole between start and goal. Small hills. Many trees.",
11248956 → "Big cave forward. Reward increases when entering cave. Goal not in cave.",
999999 → "Desert. No obstacles."

Experiments and Results

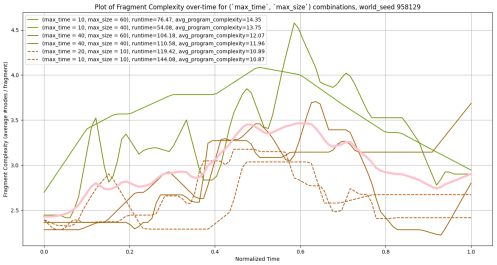
The *goal* of the experiments is to find the configuration that leads to the most complex programs and fragment pools. We define complexity by the node size of the program's syntax tree. For the experiments, we run the algorithm across multiple configurations, based on the Cartesian product of all selected values. For all the plots, two rules follow. On the y-axis, we plot the average fragment complexity. Second, the program complexity is only annotated next to the plots, or mentioned in the legend.

Experiment 1 - "Quantity" config changes

For the "quantity" configuration experiment, we tune `max_time`, the maximal time FrAngel runs before timing out, in seconds, and `max_size`, the maximal size of generated programs, in the number of nodes. The **hypothesis** is that larger sizes and larger times will increase complexity. For the latter, we presume that larger times allow for more *exploration*, and thus possibly finding larger programs.



(a) Experiment 1 - world 6354



(b) Experiment 1 - world 958129

Figure 8: Results of Experiment 1 - running each configuration 3 times. Selected few combinations are shown, to demonstrate trends. On the x-axis, we have the normalized time, from starting the synthesis to completing the task (or timing out).

Let us first discuss a trend present across all experiments. The complexity of fragments peaks in the middle of execution, shown with the thick *pink* trend lines. Initially, we expected the most fragments to appear at the end since the algorithm would have the most time to generate them. However, checkpointing significantly impacts this, since it reconsiders its fragment set at each one. In a navigation task, when checkpointing close to the goal, changes in movement direction can drastically affect the player's position. Often, it leads to negative rewards and no mined fragments.

Regarding the results in Figure 8, the trend is clear: lower `max_time` and higher `max_size` favor fragment complexity. On the plot, we compare opposite configurations `max_time` and low `max_size`. From the two, `max_size` impacts fragment complexity more positively than `max_time`. This is shown by the difference between the yellow (midway) configs and the red (worst) ones. The best configuration had over 30% higher fragment complexity than the worst.

The effect of `max_size` is expected - limiting the size of programs necessarily limits their complexity. While the prediction was wrong that high `max_time` is better, conceptually it still has merit. Instead of running the algorithm longer for fragment *exploitation*, the environment does the exploitation for us. Importantly, fragments are not carried over between checkpoints. The context is different each time, so they would not be relevant. Thus, prolonging the search at one checkpoint does not help in the long run. Instead, starting from a new checkpoint resets the context, and new relevant fragments can be found. We can conclude that frequent checkpointing (smaller `max_time`) and larger program sizes increase the number and complexity of fragments.

Experiment 2 - "Quality" config changes

For configuring the quality of fragments, we focus on three main parameters - `use_fragment_chance`, which determines the probability of including fragments into a generated program, `use_entire_fragment_chance`, the probability to entirely replace children in with fragments, and `gen_prob_similar_new`, the chance to instead replace them with a newly generated program. We **hypothesize** that excessively exploiting entire fragments, though it makes programs more relevant for the checkpoint, will decrease their size and complexity.

We see that lower values are preferable for using fragments. The former two's top configurations have a 20% and 10% difference in fragment complexity to their worst configs, respectively. Usually, fragments are smaller than newly generated programs. However, they are generally more relevant to the behavior of the program. Similarly, starting with fragments (high `use_fragments_chance`) leaves fewer options for modifications, so the ending programs will not be as complex. As to `gen_prob_similar_new`, the opposite is observed. For this parameter, higher values tend to generate more complex fragments. We simply reverse our reasoning from earlier. Making it more likely to generate children from scratch often leads to more complex modifications. For the last feature, however, the results are not as definitive. For instance, the program complexity is mostly the same across configurations.

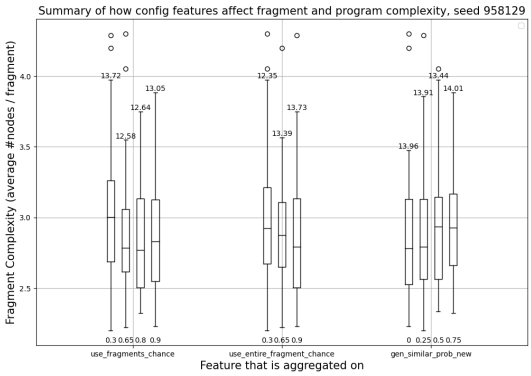


Figure 9: Experiment 2 - running each combination of features 5 times, on world seed 958129, and then aggregating the results by feature value. Here, we aggregate results based on the modified feature. On the x-axis, for each feature, we list the tested values.

To summarize, decreasing the use of fragments will increase the size and complexity of programs. Smaller fragments are simply more likely to be chosen. Each program with subtrees of height `n` will have at least as many `n - 1` sized subtrees - its direct descendants. Since most fragments are 2- or 3-sized, the total complexity will be reduced if we excessively exploit them. The trade-off here is that we are sacrificing "program relevance" by not using fragments. In other words, we sacrifice *exploitation* for *exploration* of larger replacements. This trade-off between complexity and relevance is important for understanding the results.

Experiment 3 - Inverting fragment mining condition

For this experiment, we compare flipping the *remember* condition to store more complex programs, instead of simpler ones. Furthermore, we compare this change for the original FrAngel config and a tuned version of the algorithm, based on the observations from the previous two experiments. Our tuned model uses the following values: `max_size = 60`, `max_time = 10`, `use_fragment_chance = 0.3`, `use_entire_fragment_chance = 0.3`. `gen_similar_prob_new` is ignored, since the results were not definitive in the second experiment. The goal is to see if aggregating all performing features individually also performs better. Furthermore, we want to know if flipping the condition aids program complexity. We **hypothesize** that this is indeed the case, for both subquestions.

Firstly, note how the tuned version significantly outperforms the original configuration by fragment complexities, easily averaging a 25% improvement overall, and outperforms it by fragment complexities on all world seeds. Next, note how the inverted configurations have much higher average fragment complexity. In most seeds, it is around a 2 node or 60% difference.

Naturally, larger stored programs imply larger fragments mined. This change forcefully increases the average fragment size during *exploitation*. Note, however, that the "complex" versions do not have statistically significant improvement in program complexity. The inversion of the condition

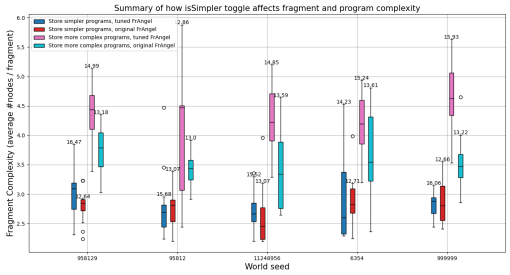


Figure 10: Experiment 3 - running each combination 12 times, for each world seed.

only affects the mining process by implementation, after all. We conclude that the tuned aggregated version and flipping the *remember* condition indeed perform significantly better (for the latter, only *fragment* complexity).

Experiment 4 - Recursive rule preference

For the last experiment, we look into giving preference to recursive rules. Recursive rules map a single statement to multiple other statements, e.g. `:(Statement = Statement ; Statement)`. Here, instead of forcing a complex *exploitation* process like the last experiment, we focus on complex *exploration*. Here, we just add extra recursive rules to the grammar. We define a "recursive depth" parameter, which is the number of `Statements` on the right-hand side. We **hypothesize** that the programs and fragments will be more complex as a result of the recursive rules.

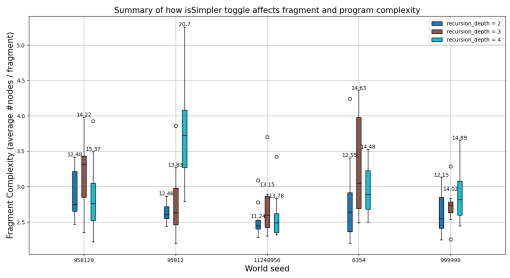


Figure 11: Experiment 4 - we consider recursive depths 2-to-4, and running each 8 times, for each world.

Yet, this is the one experiment where results are unclear. Having more recursive rules seemingly does not increase program or fragment complexity. In some worlds, a depth of 3 performs best, while in others, 4 does. There is an overperforming configuration on World 95812 (the one with recursive depth 4). However, interestingly the two versions with smaller parameters managed to solve the task within all tries, while the one only managed to solve the task once. Thus, this is an outlier that is ignored in the conclusion.

No reasonable conclusion in favor of complex subprograms can be drawn from this experiment. Presumably, it is due to the high error rate of the programs with many statements. If the program leads to a negative reward, it will not be saved for fragments, after all. Furthermore, any modifications can nullify the recursive statements.

5 Responsible Research

The research paper was written, and all supplemental coding was executed, with reproducibility in mind. This is so future researchers have an easier time contributing to the challenge of solving Minecraft with program synthesis.

The paper lays out our implementation of FrAngel in pseudo-code, with changes highlighted. The 'Herb.jl' library is open-source, with all the code publicly accessible [5]. This includes the algorithm implementation, MineRL integration, experiment setups, and the data collected. The *frangel* branches in `HerbSearch` and `HerbInterpret` contain the FrAngel implementation. All the code is documented with docs and inline comments. One may refer to the *frangel-with-minerl-exploit* branch in `HerbSearch` for the MineRL integration and experiments. The wiki page [1] under the repository contains guides for setting up 'Herb.jl', Julia, and the MineRL environment. These guides have been tested by multiple peers that work on the same module. The paper describes the exact methodology for conducting the experiments. Finally, we note that the hardware used for running the tests can significantly impact the runtimes. It is up to the reader to tune their expectations based on their testing system in comparison to ours.

6 Conclusion and Future Work

This paper looked into the novel application of program synthesis for non-zero-sum reward-based games, specifically Minecraft. The component-based program synthesizer FrAngel was adapted to accept any grammar and program generator. We model the inductive specification by splitting the reward between the goal and the player. FrAngel then selects the best partial solution to get closer to the goal. The environment uses checkpointing, splitting the task into subtasks to make the problem tractable to a synthesizer.

Experiments focused on tuning FrAngel to find complex [sub-]programs, crucial for capturing task intricacies. The first experiments examined configuration changes affecting fragment generation and usage frequency. Configurations favoring fewer fragments and more program generation yielded higher complexity. Then, we looked into slight implementational changes in FrAngel. Forcing the agent to store more complex programs for fragment mining will necessarily result in more complex subprograms. However, using higher-depth recursion of statements did not improve overall complexity.

This paper only lays the foundation for Minecraft program synthesis. Yet, there are many things to be investigated. Most importantly, the algorithm was only tested on the navigation task. We strongly encourage others to test the algorithm on other tasks, like chopping trees, building, mining for metals, etc. The visual element of the environment should also be used, something that the current integration with MineRL disregards. Once that is used, the synthesizer will certainly find more general solutions, which can be reused for other tasks, and perhaps be able to solve multiple tasks in succession.

[1] You can find the wiki pages here and here.

References

[1] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, pp. 1–119, 01 2017.

[2] A. Solar-Lezama, *Program synthesis by sketching*. PhD thesis, University of California at Berkeley, USA, 2008.

[3] X. Chen, D. Song, and Y. Tian, "Latent execution for neural program synthesis," 2021.

[4] K. Shi, J. Steinhardt, and P. Liang, "Frangel: Component-based synthesis with control structures," *CoRR*, vol. abs/1811.05175, 2018.

[5] T. Hinnerichs and S. Dumancic, "Herb.jl: A library for defining and efficiently solving program synthesis tasks in julia," 2024. GitHub repository.

[6] G. N. Yannakakis and J. Togelius, "A panorama of artificial and computational intelligence in games," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 7, no. 4, pp. 317–335, 2015.

[7] J. R. Mariño and C. F. Toledo, "Evolving interpretable strategies for zero-sum games," *Applied Soft Computing*, vol. 122, p. 108860, 2022.

[8] Z. Boddy, "Minecraft crosses 300 million copies sold as it prepares to celebrate its 15th anniversary," 10 2023.

[9] G. William, "Minerl: Towards ai in minecraft; minerl 0.4.0 documentation — minerl.readthedocs.io." https://minerl.readthedocs.io/en/latest/index.html, 2020. [Accessed 03-06-2024].

[10] A. N. Kolmogorov, "Zur deutung der intuitionistischen logik," *Mathematische Zeitschrift, 35, 58-65*, 1932. (Engl. Transl. in Mancosu P. (Ed.), pp. 328-334).

[11] A. W. Biermann, "The inference of regular lisp programs from examples," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 8, pp. 585–600, 1978.

[12] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, (New York, NY, USA), p. 317–330, Association for Computing Machinery, 2011.

[13] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *2013 Formal Methods in Computer-Aided Design*, pp. 1–8, 2013.

[14] S. Kensen, S. Jacob, and L. Percy, "Frangel." https://github.com/kensens/FrAngel, 2018.