Program synthesis from game rewards – finding complex subprograms for solving Minecraft

1. Intro / Background

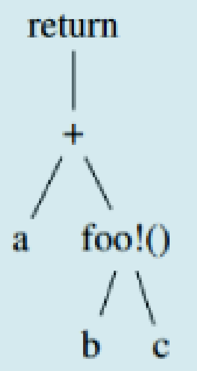
Program synthesis is the task of generating program to solve a problem. A specification describes the requirements, and can have many forms.

Inductive program synthesis is one approach. It uses an inductive specification (I/O examples), a grammar to specify syntax, and searches enumeratively.

FrAngel – program synthesizer that uses fragments for exploitation and angelic conditions for exploration.

Fragments are useful subtrees from previous programs.

Angelic conditions generalize statements for efficient search.




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

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

Formulate program synthesis for playing games – specification from game rewards. Then, integrate this algorithm into MineRL.

Then, we tune FrAngel to find more complex subprograms for solving Minecraft.

2. Research questions

How do we define program synthesis from rewards?

How to explore game environments to discover useful actions?

How do we adjust the FrAngel program synthesizer to discover more complex subprograms?

3. Methodology

3.1 Generalizing FrAngel

Allow arbitrary grammars and generators.
Add fragments to grammar.
Angelic conditions with placeholders
Keep track of the visited space.
Store partial solutions.

3.2 Defining program synthesis from rewards

Split the reward difference between the goal and player into segments – each one is an I/O example. Higher or equal final reward passes a test.

3.3 Integration with game environment

We run each FrAngel program in MineRL, and get final reward.
Checkpointing – after every FrAngel cycle, start from the new best position.

3.4 Experiments – complex subprograms

Configuration – change FrAngel's config to pick fragments, or modify with fragments, more/less often.
Implementation – change FrAngel's implementation to favor complex programs.

AUTHOR
Alperen Guncan
A.I.Guncan@student.tudelft.nl

SUPERVISOR
Tilman Hinnerichs
t.r.hinnerichs@tudelft.nl

PROFESSOR
Sebastian Dumančic
s.dumancic@tudelft.nl

4. Experimental Setup

Our experiments are on the dense navigation task. The goal is to find a diamond block, 64 blocks away. Reward is inversely proportional to the goal. We keep track of two metrics – program and average fragment complexity.

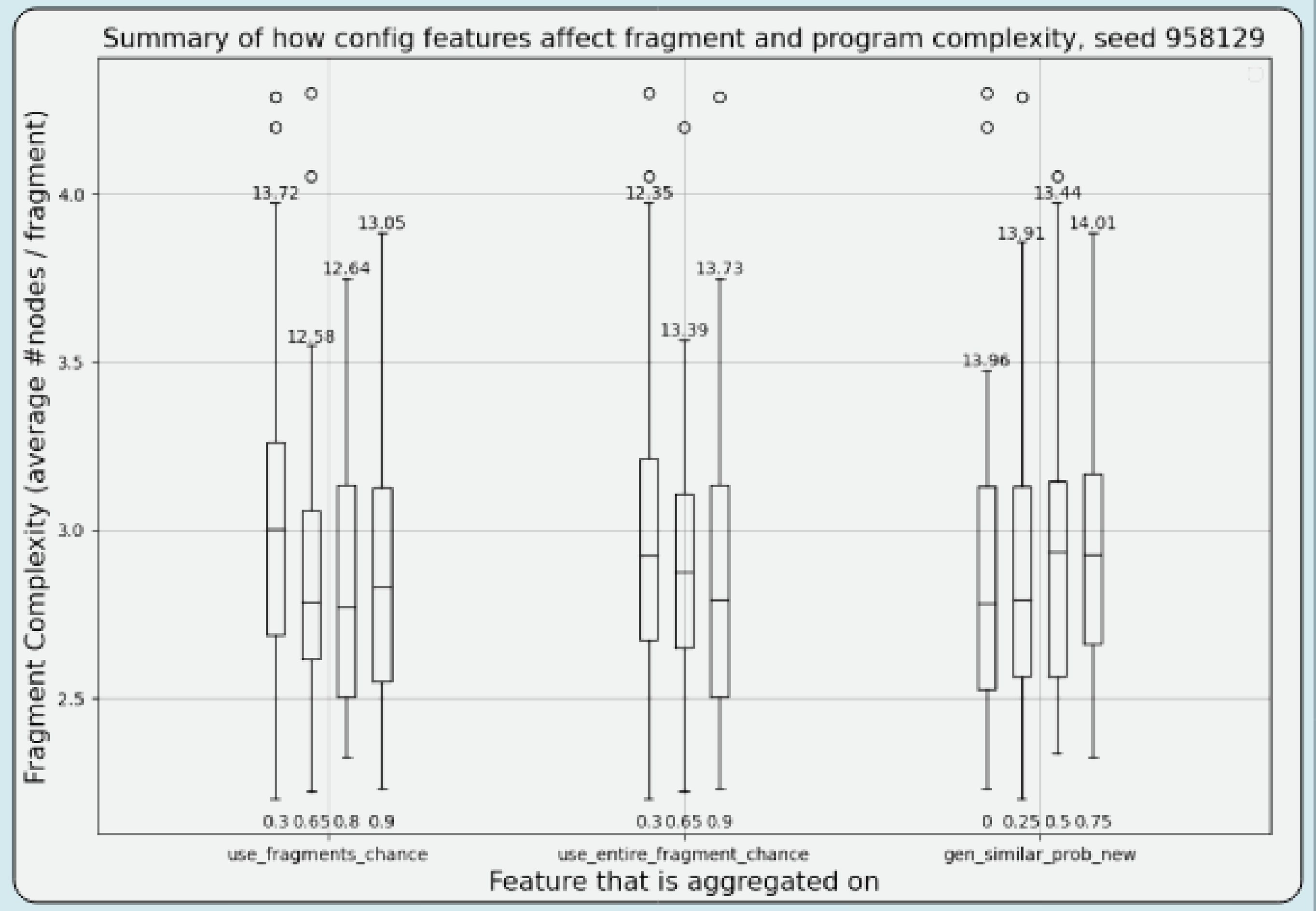
5. Results

Experiment 1 – Fragment config changes

Here, we focus on how fragment usage and rule selection affect complexity.

We tune the parameters use_fragments_chance, use_entire_fragment_chance, gen_prob_similar_new.

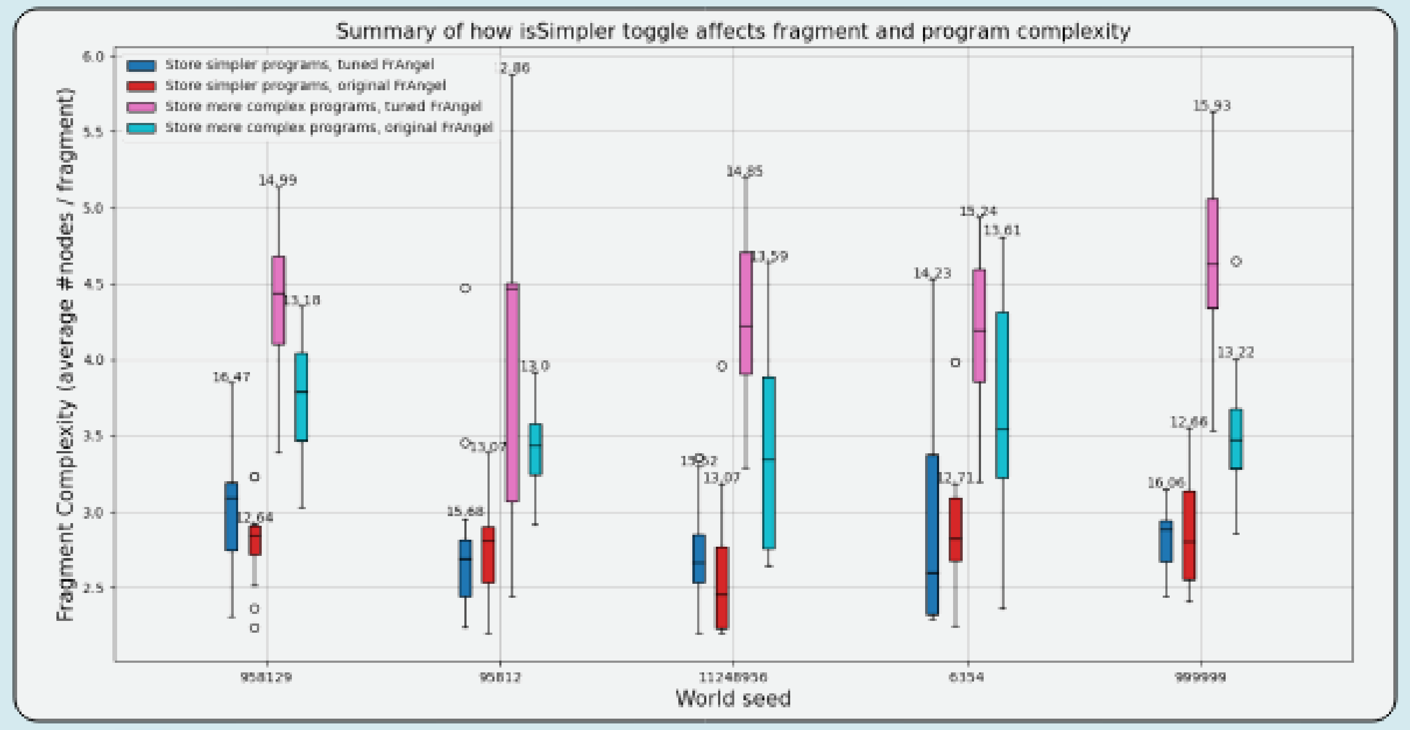
Hypothesis – more fragments means higher relevance, but less complexity.



Experiment 2 – "remember" condition

We flip the condition for remembering programs, to instead resolve ties with the more complex program.

Hypothesis – storing more complex programs leads to complex fragments



6. Conclusion

We increased average fragment complexity by reducing the probability of picking fragments for rules, and for modifying programs with fragments. This, however, trades-off program relevance. Checkpointing, however, minimizes the issue, due to context-switching. Flipping the "remember" condition also increases complexity – mining from more complex programs preserves complexity.