PROGRAM SYNTHESIS FROM REWARDS WITH PROBE

ADJUSTING PROBE TO INCREASE EXPLORATION WHEN SYNTHESISING PROGRAMS FROM REWARDS IN MINECRAFT

Author
Nils Marten Mikk
n.m.mikk@student.tudelft.nl

Responsible Professor
Sebastijan Dumančić
s.dumancic@tudelft.nl

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

1 INTRODUCTION

Program synthesis is the task of generating a program according to some user-provided specification.

Different methods of specification: I/O examples, natural language, formal specification, traces.

Explore novel method of specification: rewards.

MineRL – Python library used for interacting with Minecraft.

Synthesis might get stuck in local maxima. Try to avoid by increasing exploration.

Probe – program synthesiser that updates the probabilities of a probabilistic CFG during execution.

Guided bottom-up search – search algorithm used by Probe that enumerates programs by increasing cost level.

Partial solution – program that solves some examples.

Update function – increase probabilities of rules that appear in promising partial solutions.

2 RESEARCH QUESTIONS

How to define program synthesis from rewards?

How to adjust Probe to learn programs from rewards?

How to increase the amount of exploration and how does it affect the runtime when learning from rewards with Probe?

4 EXPERIMENTAL SETUP

Dense navigation environment: reach diamond block approximately 64 blocks from spawn. Receive reward based on how much closer to goal after each step.

Grammar: 1 action after best program, 8 directions, 6 step sizes, always sprint and jump.

Run each experiment ten times on five worlds.

3 METHODOLOGY

1. Generalise Probe.

Allow the use of arbitrary search algorithms.
Easily changeable parameters for ease of experimentation.

Use number of programs as cycle length instead of levels.
Keep track of cycle in Probe instead of in search algorithm.
Add evaluation cache, set of partial solutions to Probe.

2. Define a grammar and a method for evaluating generated programs in MineRL.

Program: sequence of (*steps*, *action*) instructions.

Evaluation: iterate over sequence and execute the *action* *steps* times.

3. Define program synthesis from rewards for MineRL.

Update grammar to use program with highest reward for first instructions in sequence. Search continues from best position.

4. Adjust Probe to learn MineRL programs from rewards.

Observational equivalence: programs end up in approximately the same position.

Partial solution: program that improves the best reward.

Selection function: select 5 programs with highest reward.

Update function: increase probabilities of last instructions. Set start to best program.

5. Find ways to increase the amount of exploration and analyse their effect on runtime.

5 EXPERIMENTS & RESULTS

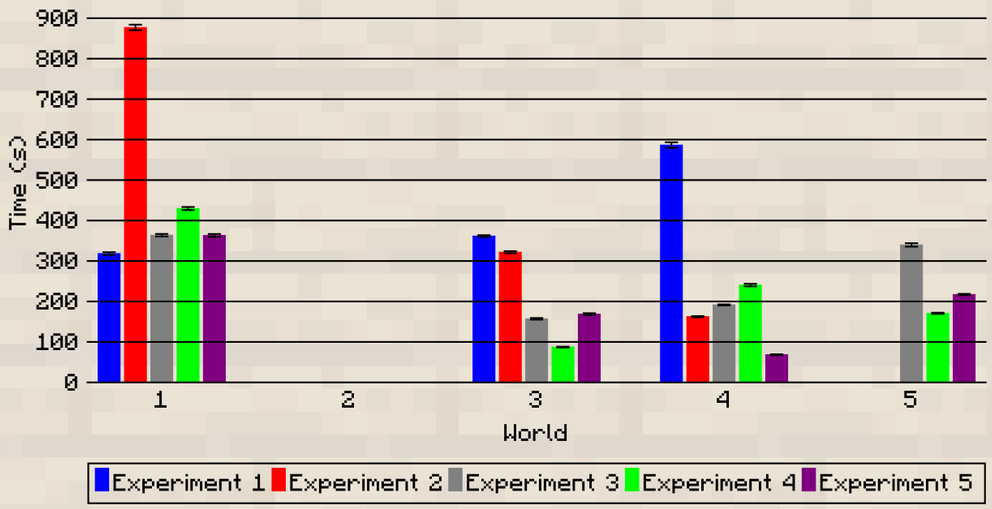

Figure 1. Average runtimes of experiments 1–5 over ten runs in seconds. A missing bar means that the experiment timed out for at least one run.

Experiment 1: Baseline. Unable to solve worlds 2 and 5, gets stuck in local maxima.

Experiment 2: Attempt to get out of cave in world 5 by allowing multiple actions after best program. Did not solve world 5. Changed runtimes of other worlds even though they do not use multiple actions after the best program.

Experiment 3: Enumerate different directions before different step sizes to find right direction faster. Can solve world 5.

Experiment 6: Randomise probabilities after each cycle. Able to solve world 2 in two runs out of ten. Tends to be more successful in worlds with fewer obstacles.

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

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

6 CONCLUSIONS

Use program with best reward as first instructions to guide the search.

Redefine partial solutions, observational equivalence, and update function to learn programs from rewards with Probe.

Increase exploration by changing grammar, cycle length, update function.

By increasing exploration it is possible to avoid local maxima and solve more environments.

Depending on the environment, increasing exploration can either increase or decrease runtime.