Author: Timur Mukminov
Contact: t.imukminov@student.tudelft.nl

Program Synthesis in Minecraft

Supervisor: Sebastijan Dumancic
Tilman Hinnerichs

Defining Rewards and Exploiting Successful Programs

1 Background

Page 1

Program Synthesis – Technique that creates programs that provably satisfy a given high-level formal specification.

Probe – A synthesis algorithm that uses a probabilistic context free grammar (PCFG) which gets updated after every cycle

MineRL – A python library which uses Minecraft to provide tasks and environments.

2 Objective

Page 2

How can we modify probe to use rewards to synthesis programs?

How do explore game environments to discover useful actions?

How do we adjust the Probe program synthesiser to exploit successful programs it has already discovered

3 Methodology

Page 3

Generalise Probe – Modularised the algorithm to allow the interchange and use of custom search function, PCFG, selection of partial solutions and updating grammar methods.

Selection function – Selects partial solutions which obtain the largest reward by traveling the closest to the desired destination

Observational equivalence – After evaluation some programs have same result. Therefore we store the evaluation result and if a program has the same evaluation, discard it.

Update grammar – Uses the algorithm below to change the probabilities of a grammar rule appearing, tailoring the search function to find the final solution

$$p(R) = \frac{p_u(R)^{(1-Fit)}}{Z}$$

p_u is the uniform probability
Z is the normalisation factor
Fit is a value that ranges from 0 to 1

Integrate with MineRL – set up an environemnt where agent must travel to destination 64 blocks away. The closer the agent gets, the larger the feedback reward

4 Set-Up & Results

The experiments were run 3 times in 5 diverse world seeds Most experiments rely on changing the fitness based on reward for the grammar rule.

Experiment 1 – Constant Fitness = 0.3. Able to solve 1 world and nearly solves another.

Experiment 2 – Linear Fitness = reward/100 Able to solve 3 worlds. Vastly lower amount of time required. Sometimes skips the 95% threshold and reaches goal directly like in fig. 5

Experiment 3 – Exponential Fitness = 1- exp((reward/55)^3 This fitness exploited a lot at all stages therefore wasn't able to navigate a dense forest in world seed 6354

Experiment 4 – Logarithmic Fitness = log10((1+reward)/2). Surprisingly did well on the harder of the 3 worlds.

Experiment 5 – Complex fitness using 2 variables: reward and number of occurrences of grammar rule in partial solutions. Very exploitative of the partial solutions and performed the best when reaching 95% of the reward however struggled on some worlds to reach the final goal.

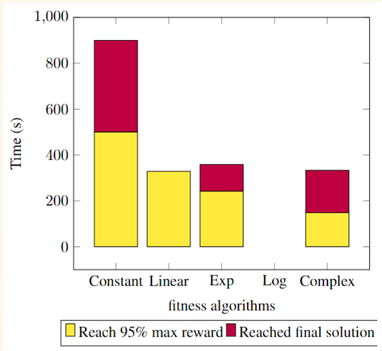

Figure 4: World 1. Time to find solution using different fitness algorithms. Missing Bars means no solution

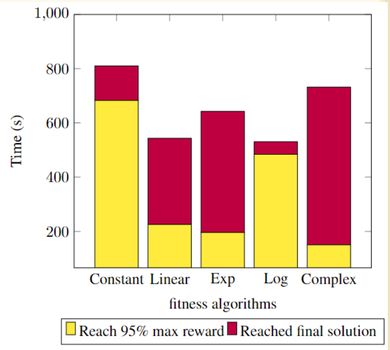

Figure 5: World 2. Time to find solution using different fitness algorithms

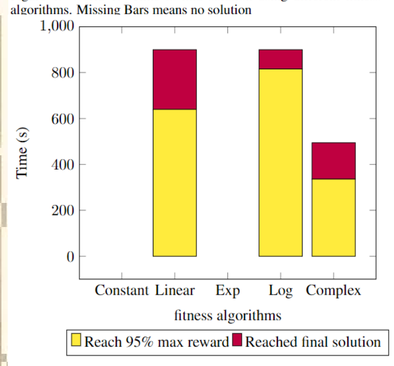

Figure 6: World 3. Time to find solution using different fitness algorithms. Missing Bars means no solution

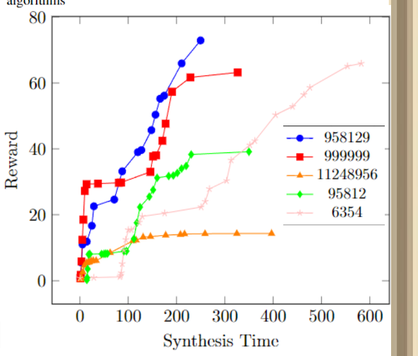

Figure 12: fit = best_reward/100

Page4

Experiment 7 – The same fitness was used as in Experiment 1-5 however this time the initial probabilities of the grammars were set to a successful or partially successful grammar for that world.

- Grammar more tailored to final solution
- Time till 95% reward decreased
- Time to reach full reward increased
- Some worlds became unsolvable – not enough exploration
- Different starting grammar – different set of worlds could be solved

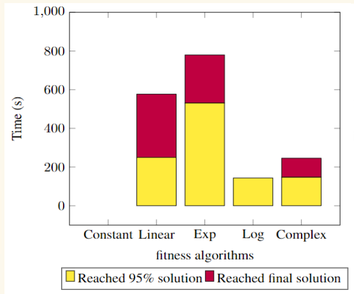

Figure 9: World 1. Time to find solution using different fitness algorithms and with starting probabilities

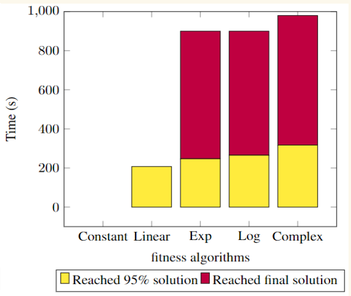

Figure 7: World 2. Time to find solution using different fitness algorithms and with starting probabilities

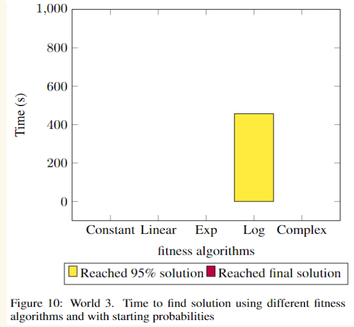

Figure 10: World 3. Time to find solution using different fitness algorithms and with starting probabilities

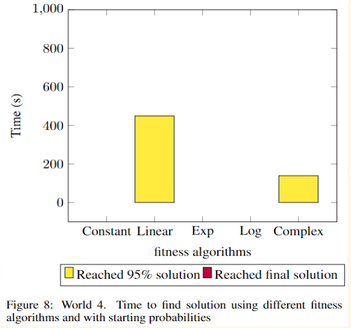

Figure 8: World 4. Time to find solution using different fitness algorithms and with starting probabilities

5 Conclusion

Page 5

- Increase exploitation by changing starting grammar probabilities
- Use different fitness algorithms to change how much probabilities change
- Increasing exploitation reduces time to almost solve but increases time to actually solve
- Too much exploitation kills off exploration meaning it is difficult to reach destination when near.
- Speedup depends on environment: simpler worlds show a significant speedup, more complex and difficult worlds

Starting with an exploited grammar reduces initial time to locate correct direction for travel and allows to take a different route to goal. However exploration is lessened even more and it now often doesn't even find a solution