Three Games-Related Benchmark Problems for Program Synthesis

Summer 2021 Project Report

Olivier Vadiavaloo

Supervisor: Levi H.S. Lelis

University of Alberta

Abstract

In this report, I present a domain-specific language (DSL) for synthesizing strategies for playing one of three different games: Pong, Flappy Bird and Catcher. I also report on the results of employing a synthesizer, that uses stochastic search, to probe the program space, defined by the DSL, for strong strategies. Since discovered strategies often make use of constants within various operations, the synthesizer uses a search algorithm to find a strategy's structure and then, relies on an optimizer to find those constants that help the strategy achieve the best score. In an attempt to speed up the synthesis process, I also used the concept of triage during evaluation and optimization, leading to six major configurations for the synthesizer. I conducted experiments for each configuration on all three domains and the results show that in one domain, it is better to avoid using the optimizer and in the other two, using the optimizer is important for finding strong strategies. They also show that configurations using triage find strong strategies faster than their non-triage counterparts.

1 Introduction

In program synthesis, the task is to generate a computer program that satisfies the user's intent (1), which is usually specified through logical expressions, input-output examples (2) or a program sketch (5). In the context of games, the task is to find programmatic strategies for playing the game. Thus, the user's intent is the game's specification.

Normally, in program synthesis, one also defines a domain-specific language (DSL) that is expressive enough for generating strategies that can solve the problem. The DSL must also constrain the strategy space such that finding a solution to the problem is feasible in practice. Therefore, I designed a DSL for Pong, Catcher and Flappy Bird based on my own strategies for the domains and expressed its syntax using a context-free grammar (CFG). Strategies for any of the three domains can then be represented using this DSL.

Any implementation of Pong, Catcher and Flappy Bird, together with the DSL, form three games-related benchmark problems for program synthesis. I attempted to find solutions to these problems using a synthesizer that employs stochastic search and an optimizer. The synthesizer also uses a triage scheme to speed up the search by focusing on the evaluation and the optimization of promising strategies.

As an overview, I will explain the DSL used to synthesize programmatic strategies for each domain, elaborate on how my synthesizer functions, describe the experiments that I conducted for each domain and report the results. I will then discuss the findings in the last section.

2 Games

Pong, Flappy Bird and Catcher form three benchmark problems for synthesizing strategies for games. All three domains fall under the arcade genre and have very simple rules and goals. In this project, I used the implementations of Pong, Flappy Bird and Catcher from the Pygame-Learning-Environment (6), or PLE for short. The implementations found in PLE allow easy access to the game state and can compute a reward based on the strategy's performance. In this work, I directly used the reward returned by the game objects from PLE as the measure of a strategy's strength during its evaluation.

2.1 Catcher

In catcher, an agent controls a rectangular paddle that it can move either left or right. As the game begins, squares spawn randomly above the agent's paddle and fall to the bottom of the game's window. The agent must move the paddle such that it intersects, or catches, the falling object. The agent loses and the game stops if it is unable to catch three falling objects.

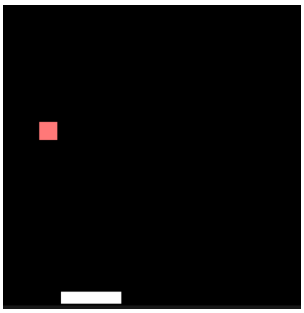

Figure 2.1: A frame from a game of Catcher, with the agent controlling the bottom rectangular, white paddle.

2.2 Pong

Pong is a classic 2-player Atari game where two agents control a rectangular paddle each and a circular object, representing a ball, changes direction when intersecting with one of the paddles. If the ball touches the left edge of the window, the agent controlling the right paddle wins a point and similarly, if it touches the right edge, the left paddle's agent wins a point. The agent which reaches 20 points first wins the game. In PLE, the implementation for Pong also provides a strong strategy that acts as an opponent against which I evaluated all synthesized strategies.

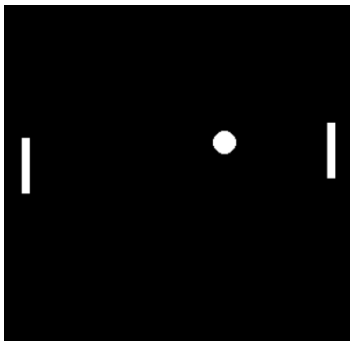

Figure 2.2: A frame from a game of Pong, with the agent controlling the left paddle.

2.3 Flappy Bird

Flappy Bird was once a very popular mobile game where the agent controls a sprite that has to pass through gaps between two pipes of varying lengths throughout the game. If the sprite intersects with one of the pipes or a window edge, the agent loses and the game stops. The agent can either allow the sprite to fall or move the sprite up so that it passes through a gap.

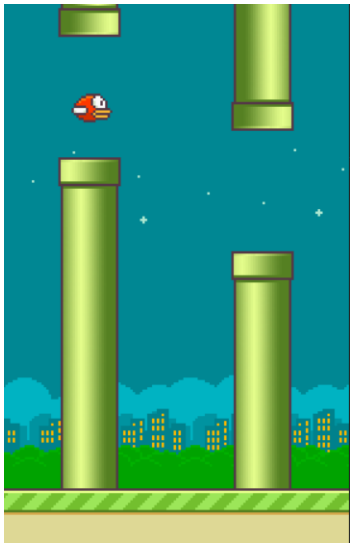

Figure 2.3: A frame from a game of Flappy Bird.

3 Domain-Specific Language

$$
\begin{aligned}
&I \rightarrow S_1 \mid S_2 \\
&S_1 \rightarrow if\ (B)\ then\ \{R\}\ S_1 \mid R \mid \epsilon \\
&S_2 \rightarrow if\ (B)\ then\ \{S_1\}\ else\ \{S_1\} \\
&D \rightarrow\ D_1 \mid D_2 \\
&D_1 \rightarrow\ T \mid c_1 \mid c_2 \mid c_3 \mid ... \mid c_n \\
&D_2 \rightarrow\ D_1 + D_1 \mid D_1 - D_1 \mid D_1 * D_1 \mid D_1 / D_1 \\
&T \rightarrow 0.01 \mid 0.02 \mid 1.00 \mid ... \mid 100.00 \\
&B \rightarrow D < D \mid D > D \mid D == D \mid b_1 \mid ... \mid b_m \\
&A \rightarrow actions \\
&R \rightarrow return\ A[0] \mid return\ A[1] \mid return\ A[2]
\end{aligned}
$$

I define a DSL for Pong, Flappy Bird and Catcher using a context-free grammar (CFG) $G = (V, \Sigma, P, I)$ where,

- $V = \{I, S_1, S_2, D, D_1, D_2, T, B, A, R\}$,
- $\Sigma = \{if, then, else, \epsilon, c1, c2, ..., c_n, b_1, b_2, ..., b_m, 0.01, 0.02, 1.00, ..., 100.00, return\}$,
- P is the set of relations and
- I is the initial symbol.

V is the set of non-terminal symbols and Σ is the set of terminal symbols. I is the initial, or starting, symbol of the DSL.

S_1 adds **non-nested** if statements to the DSL and potentially multiple if statements followed by a return statement. For example,

```
if (b1) then
   return A[0]
if (b2) then
   return A[1]
return A[2]
```

S_2 adds potentially nested if-then-else statements to the DSL. D is a non-terminal symbol on which two relations can be applied: one to obtain D_1 and the other to obtain D_2.

D_1 and D_2 are also non-terminal symbols on which relations can also be applied. D_1 allows strategies to access n high-level functions. For example, in Catcher, $n = 2$ and so, synthesized strategies can access two functions, PlayerPosition and NonPlayerObjectPosition. Some functions are also accessible in all three domains, such as PlayerPosition. D_1 also allows access to the non-terminal symbol, T. D_2 gives strategies access to basic arithmetic operators.

The T symbol adds constants which range from 0.01 to 100.00 to the DSL. Strategies can also use the non-terminal symbol, B, to access comparison operators and some functions that return only boolean values (e.g NonPlayerObjectApproaching). A simply gives access to an array of *actions* and R is used to represent returning an action from an array.

4 Synthesizer

Various algorithms exist to search the program space for strategies meeting the user's intent. One such method is stochastic search. The synthesizer which I used to conduct experiments implemented Simulated Annealing (3), which is a stochastic search algorithm that uses a temperature value to discard or accept solutions to probe from next in the search space. The synthesizer also uses an optimizer to find a set of constants that maximize the strategy's score. It also implements a triage scheme which it can use to focus on fully evaluating and optimizing only promising strategies.

4.1 Simulated Annealing

The synthesizer starts the search by generating a random program using the symbols from the DSL and then searches the program space by randomly mutating the program. Mutating a program can be easily done by changing a randomly selected non-terminal symbol in the abstract-syntax tree (AST) representing the program. An evaluation function, $Eval(s)$, then evaluates each strategy s on a certain number of matches for the domain for which it has been synthesized. $Eval(s)$ returns a possibly negative score for each s that it evaluates. If $Eval(s)$ is larger than $Eval(\alpha)$, where α is the current best strategy, then α is replaced by s. When $Eval(s)$ is less than $Eval(\alpha)$, the algorithm can either (a) continue the search from the previous strategy s^* that was evaluated or (b) continue the search from strategy s. If $Eval(s^*)$ is less than $Eval(s)$, then the algorithm chooses option (b). Otherwise, the algorithm relies on an acceptance function and a temperature value to decide.

4.2 Optimizer

The DSL permits synthesized strategies for all three domains to use constants ranging from 0.01 to 100.00. Depending on the constants being utilized by a strategy, its performance can be better or worse. Ideally, we would like to find those constants that maximize a given strategy's scores. This is akin to a black-box optimization problem, where $Eval$ is the black-box function, and so, a Bayesian optimizer can be used. In my project, the synthesizer finds the structure of the strategies with Simulated Annealing and then, uses a python implementation of a Bayesian optimizer (4) to find the constants that maximize their scores. The surrogate function used by the optimizer is a Gaussian Process (GP) and the acquisition function returns the constant values that yield the maximum upper-confidence bound (UCB) from the surrogate function.

4.3 Triage

If the synthesizer is configured to use triage and according to some criterion, a strategy s is believed to be worst than the current best strategy α, then s will not be evaluated on all matches. This is done to avoid wasting computational resources on evaluating strategies deemed worst than α. I also applied the triage scheme during optimization to focus on optimizing promising strategies only. In section 6, we see that experimental results for two of the domains support the use of this configuration during optimization.

5 Experimental Design

5.1 Synthesizer Configurations

The synthesizer has 6 main configurations since both the optimizer and the triage scheme can be incorporated in the strategy search only if desired. I executed the synthesizer with each of the following configurations 10 times in each domain and compared their average scores (see 6.1).

- Synthesizer without optimization and without triage evaluation (Nop)
- Synthesizer without optimization and with triage evaluation (NopTe)
- Synthesizer with non-triage optimization and without triage evaluation (NTop)
- Synthesizer with non-triage optimization and with triage evaluation (NTopTe)
- Synthesizer with triage optimization and without triage evaluation (Top)
- Synthesizer with triage optimization and with triage evaluation (TopTe)

5.2 Running the Synthesizer

Since the synthesizer uses Simulated Annealing, an initial temperature, final temperature and an temperature-decay function must be provided. The initial and final temperatures were 1 and 2000 respectively in all experiments. The temperature-decay function was a simple rational function of the form:

$$\frac{T_c}{1+\alpha i}$$

T_c is the temperature value after iteration i of Simulated Annealing and α is an arbitrary value that was set to 0.9 in all experiments.

For those experiments with configurations that use the triage scheme, a confidence value must be specified to decide if a strategy is promising enough for continuing the evaluation or optimization. In this work, I ran the experiments with a confidence value of 95% for each domain.

Each configuration was run 10 times for every domain on Compute Canada's Beluga cluster. For Catcher, the synthesizer had a time budget of 12 hours and 16 CPUs, with 1000MB per CPUs. For Pong and Flappy Bird, the time budget was 10 hours and 6 CPUs were used, with 1000MB per CPU. This leads to a total of 180 experiments.

6 Results and Discussion

During each experiment, the synthesizer stores the score of each new best strategy. I then plotted the average curve of the stored scores against running time for each configuration in all three domains. While the strategies can also be assessed by how explainable they are, I will assess them mainly through their scores in their respective domains. Strong strategies will have larger scores than weaker strategies and it is possible for a strategy to have a negative score.

6.1 Catcher

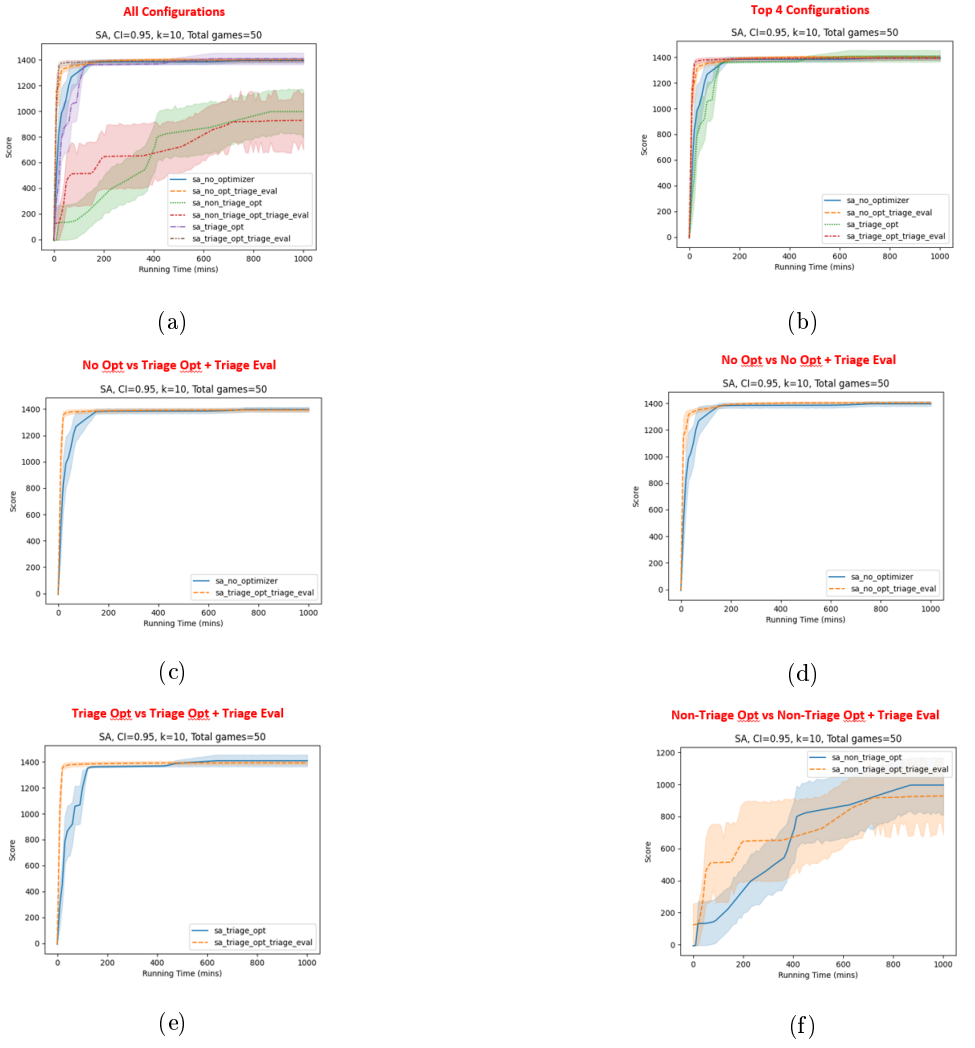

Figure 6.1: Average Curves of best scores against running time for the experiments run for Catcher

The average curves for Catcher indicate that using the synthesizer with the TopTe configuration yields the best strategies. Using the NopTe configuration provide strategies with scores close to the ones for TopTe on average. On the other hand, with NTopTe or NTop, the synthesizer finds worse programs since optimization without triage consumes a considerable amount of the allocated time budget. A significant amount of time might be spent on trying to optimize strategies with bad structures for playing the game. With triage optimization, the optimizer focuses only on strategies with promising structures found by Simulated Annealing. If an optimizer is used when synthesizing strategies for Catcher, triage must be used. Otherwise, it might be better to simply use Nop or NopTe.

It is also worth noting that every configuration using triage produces better strategies quicker in the search than their non-triage counterparts. For instance, figure 6.1(d) shows that, on average, a synthesizer using Nop takes nearly 100 more minutes than a synthesizer using NopTe to find a strategy with a score of approximately 1300. Figure 6.1(e) indicates that, on average, a synthesizer with the Top configuration also takes nearly 100 more minutes than a synthesizer with the TopTe configuration to find a strategy with a score of about 1300.

6.2 Pong

For Pong, using the Nop and NopTe configurations with the synthesizer turns out to produce better scripts than other configurations on average. Using the optimizer, with and without triage, yielded worse strategies than when running the synthesizer with Nop or NopTe. This might be because constant values might not play a major role in the strategies' strengths in Pong. With NTop, NTopTe, Top and TopTe, the synthesizer spends more time on optimizing strategies at the expense of searching for a better strategy structure with Simulated Annealing. However, configurations using triage again produce better scripts earlier in the search than their non-triage counterparts on average, as evidenced by the positions of the average curves in figures 6.2(b), (d), (e) and (f).

Figure 6.2(b) shows that when using Nop, the synthesizer takes about 50 more minutes than with NopTe to produce a strategy with a score of 20 on average. In figure 6.2(d), the average curves indicate that even after 600 minutes, the strategies produced with Top have a weaker score than those produced with TopTe.

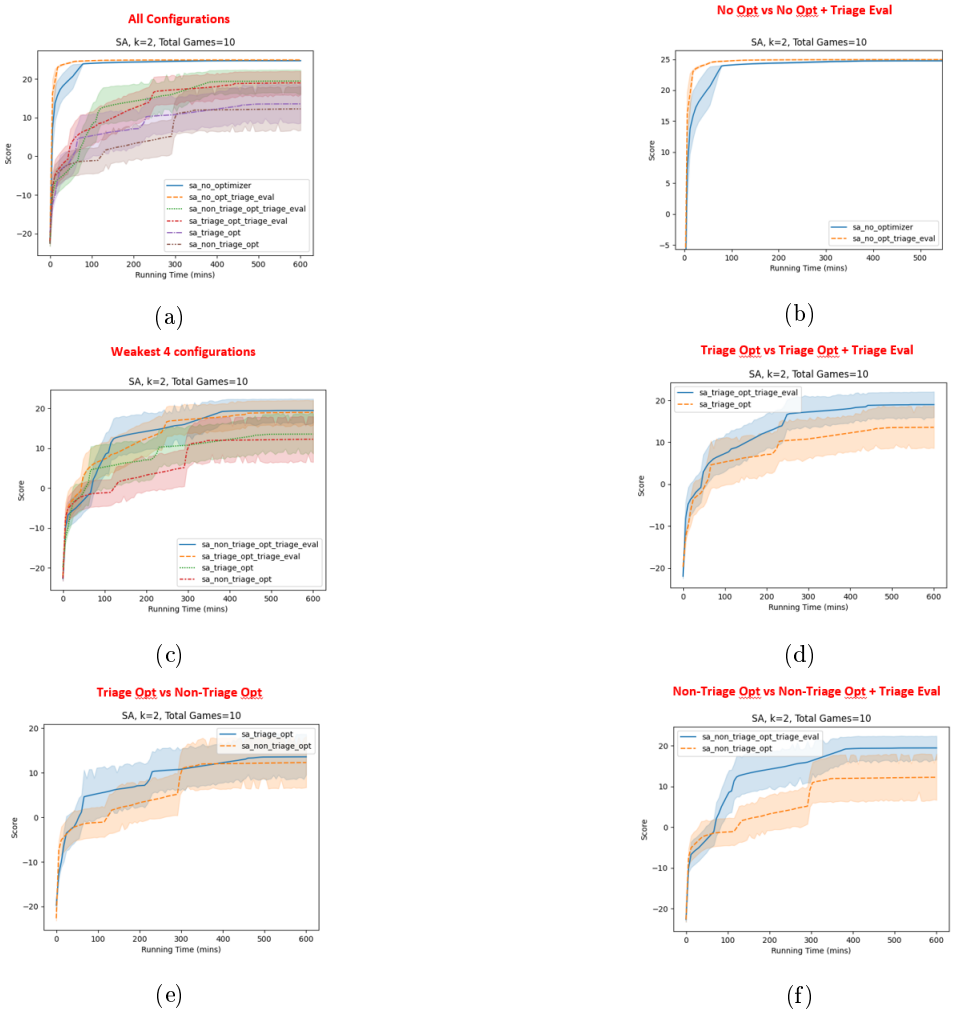

Figure 6.2: Average Curves of best scores against running time for the experiments run for Pong

6.3 Flappy Bird

In contrast to Pong, the best configurations in Flappy Bird are TopTe and NTopTe, which both use an optimizer. This might indicate that, in Flappy Bird, constant values play a more important role in a strategy's strength than in Pong. Using TopTe produces better strategies than with NTopTe on average, which might be because using triage during optimization saves time which can be used to find strategies with better structures. Applying triage during evaluation also seems to be important if the optimizer is used, as shown by the positions of the average curves for Top and TopTe in figure 6.3(e) and for NTop and NTopTe in figure 6.3(f).

Figures 6.3(b), (d) and (f) also suggests that configurations using triage again yield stronger strategies in less time than their non-triage counterparts.

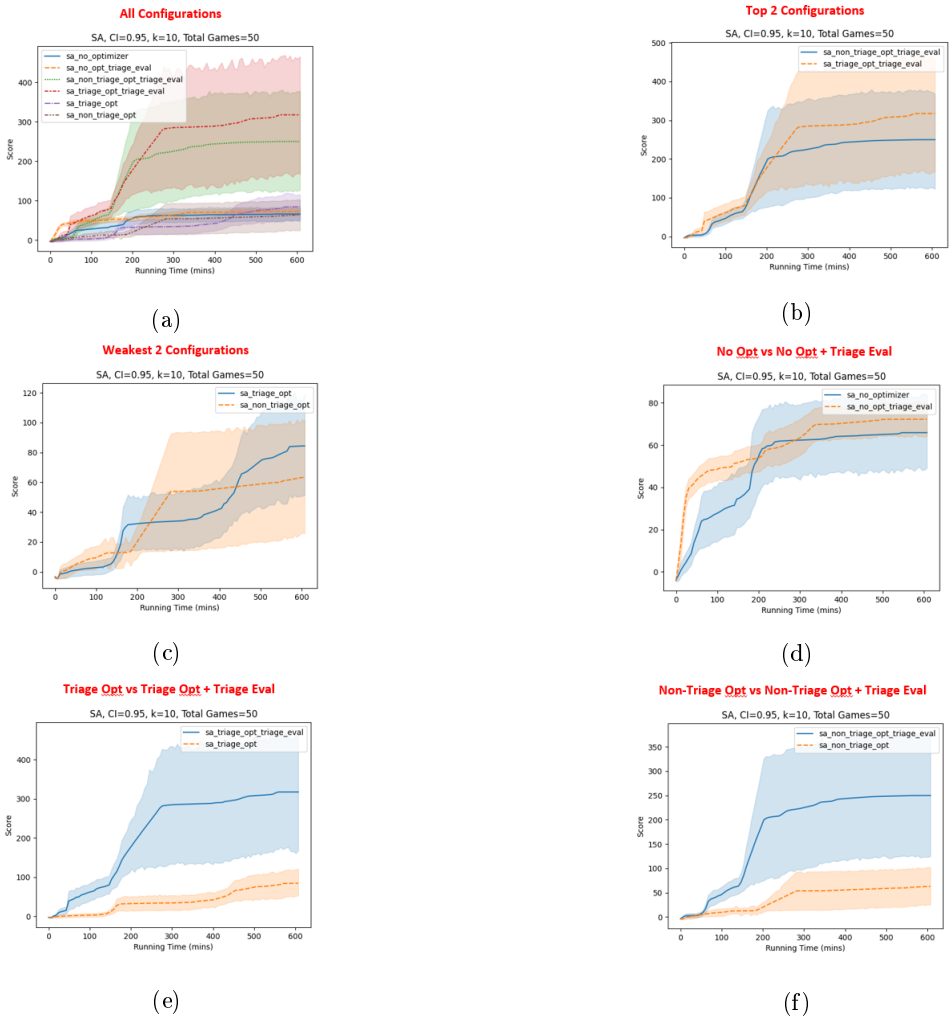

Figure 6.3: Average Curves of best scores against running time for the experiments run for Flappy Bird

Overall, the results indicate that in Pong, not relying on the optimizer produces better strategies, while in Catcher and Flappy Bird, using TopTe gives better strategies. We also saw that, in all three domains, using triage gave us stronger strategies faster during the synthesis than when performing evaluation and optimization without triage.

7 Conclusion

In this work, I presented a DSL for synthesizing strategies for playing Pong, Catcher or Flappy Bird. The DSL allows for if-then's, if-then-else's, arithmetic operations, comparison operations and different functions specific to the domain for which a strategy is being synthesized. I built a synthesizer that uses Simulated Annealing to sample the program space defined by the DSL for strategies. It can also use a Bayesian optimizer to find constant values that maximize a strategy's score. It also implements an optional triage scheme to focus on evaluating and optimizing promising strategies only, hence saving time that can be used to sample better strategies from the program space. These two features allow the synthesizer to have six main configurations.

I ran ten experiments for each configuration on every domain and plotted the average curves of the best scores against running time for each configuration in every domain. The results indicated that in Catcher and Flappy Bird, using the triage optimization and evaluation yields better strategies on average, while in Pong, it is better to avoid using the optimizer. The results also showed that in all three domains, configurations that used triage, during evaluation, optimization or both, produced stronger strategies earlier during the experiment than their non-triage counterparts.

In the future, more experiments could be run on these three domains which can serve as benchmark problems for program synthesis. We could change some of the experiment variables, such as the type of temperature decay function, the confidence value used by triage and the number of CPUs, and examine the effects on the resulting synthesized strategies and their strengths. For instance, we could use less CPUs to make the problem harder for the synthesizer or use a smaller confidence value and observe how it affects triage.

Those three domains can then serve as testing playgrounds for synthesizers with different configurations and algorithms which could lead to a better understanding of those synthesizers and hence, the creation of better tools for solving other games-related program synthesis problems.

8 Code

The code can be found at:

`https://github.com/lelis-research/PyGames-synthesis.git`

9 References

[1] Sumit Gulwani. Dimensions in program synthesis. *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming - PPDP 10*, 2010.

[2] Sumit Gulwani. Programming by examples. *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, 2017.

[3] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[4] Fernando Nogueira. Bayesian Optimization: Open source constrained global optimization tool for Python. `https://github.com/fmfn/BayesianOptimization`, 2014.

[5] Armando Solar-Lezama. The sketching approach to program synthesis. *Programming Languages and Systems Lecture Notes in Computer Science*, page 4–13, 2009.

[6] Norman Tasfi. Pygame learning environment. `https://github.com/ntasfi/PyGame-Learning-Environment`, 2016.