

Program Optimization with Local Search

by

Fatemeh Abdollahi

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science
University of Alberta

© Fatemeh Abdollahi, 2022

Abstract

Recent advancements in large language models and program synthesis have enabled the development of powerful programming assistance tools. These tools are designed to help the programmer while writing a program in an online setting. In this thesis we introduce a programming assistant that can optimize a compilable version of a program with respect to a comparable objective.

We propose a local search approach for improving programs written by humans using bottom-up search algorithms. We present Program Optimization with Locally Improving Search (POLIS), which exploits the structure of a program, defined by its lines, in order to split the optimization into small program synthesis tasks that can be solved by the existing synthesis algorithms. POLIS iterates over lines of the program, trying to improve a single line at a time, while keeping the other lines fixed. It continues to iterate until it is unable to improve the objective value of the program or runs out of time.

Our hypothesis is that humans are capable of thinking abstractly and at a high level but they sometimes miss some details of their program. Computer agents, on the other hand, can figure out the details of a program efficiently, whether it is to optimize numerical values in a program or synthesize programs for programming tasks that can be solved with short programs. We leverage this ability of program synthesis algorithms and use them to optimize each line of a program iteratively. Thus, we optimize the human-written program by optimizing each line of it.

We evaluated POLIS in a 27-person user study where the participants wrote programs for playing two single-agent games: Lunar Lander and Highway. In this study,

participants were rewarded based on their programs' score. Results demonstrate that while human programmers may be capable of writing effective high-level program structures, they often fail to optimize important details of the programs. POLIS substantially improved the quality of all programs evaluated while preserving the high-level structure the programmers designed. These results show that this approach could be used as a helpful programming assistant for problems with measurable and comparable objectives.

Preface

This thesis is an original work by Fatemeh Abdollahi. The research project, of which this thesis is a part, received research ethics approval from the University of Alberta Research Ethics Board 2, Project Name “Human-AI Collaboration for Synthesizing Programmatic Policies”, No. Pro00113586.

Acknowledgements

First and foremost, I would like to extend my sincere thanks to my advisors, Levi Lelis and Matthew E. Taylor, for their invaluable supervision, support and encouragement during my graduate studies. I would also like to thank Antonie Bodley, my writing coach, for all of her feedback on this thesis. Finally, I would like to express my very deepest gratitude to my parents and to my husband, for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Table of Contents

1	Introduction	1
2	Related Work	4
2.1	Program Synthesis	4
2.2	Program Enhancement	5
2.3	Intelligent Programming Assistant	6
2.4	Interpretable Reinforcement Learning	7
3	Background	9
3.1	Problem Definition	9
3.2	Bottom-Up Search	10
3.3	Probabilistic Guided-Bottom-Up Search (Probe)	12
4	Program Optimization with Locally Improving Search (POLIS)	16
4.1	Program Definition	16
4.2	POLIS: a Programming Assistant	16
4.2.1	Local Search	17
4.2.2	BUS for Optimization Problems	17
4.2.3	Probe for Optimization Problems	19
5	Experiments	23
5.1	Problem Domains	23
5.1.1	Lunar Lander	23
5.1.2	Highway	24
5.2	Experimental Details	26
5.2.1	POLIS Domain Specific Language	27
5.2.2	Input-Output Examples	27
5.2.3	Highlights	28
5.2.4	Restarts	29
5.2.5	Bayesian Optimization	29

5.3	User Study Design	30
5.4	Results	32
5.4.1	Demographics	33
5.4.2	Computational Results	33
5.4.3	Representative Program	40
6	Conclusion and Future Work	43
	Bibliography	46
	Appendix A: Graphical User Interface	51
	Appendix B: Original Representative Program	57
	Appendix C: DQN Hyperparameters	58

List of Tables

3.1	Programs generated for Equation (3.1) from the grammar in Equation (3.2) in the order of size. The size of a program is defined as the number of AST nodes of the program. To simplify the computation, we do not consider any pruning strategies to limit the search.	11
3.2	Programs generated for Equation (3.1) from the probabilistic context-free grammar in Equation (3.2) in order of cost defined by Probe. To simplify the computation, we do not consider any pruning strategies to limit the search.	15
5.1	A summary of Participant’s performance for Lunar Lander	34
5.2	A summary of Participant’s performance for Highway	35
5.3	Game score improvements for Lunar Lander	38
5.4	Game score improvements for Highway	39

List of Figures

5.1	Lunar lander (top) and Highway (bottom)	24
5.2	In the Lunar Lander game, the state of the spaceship is determined based on eight features at each step as in Equation (5.1). There are four actions available for the player as shown in Equation (5.2)	25
5.3	According to the Open AI Gym [47] documentation, the reward after each step of the game is calculated based on these rules. The player at the end of each episode receives an additional reward of -100 or +100 points for crashing or landing safely respectively.	25
5.4	In the Highway game, each car is represented by four features as seen in Equation (5.4). At each step of the game, the player can see the green car's and the two nearest car's features as shown in Equation (5.5). Equation (5.6) indicates five actions available at each step at Highway.	26
5.5	At each step of Highway, the reward is calculated as Equation (5.7) where v is the speed of the car and $[15, 30]$ is the speed range. For v less than 15, the speed reward is 0 and for v more than 30, the reward speed is 0.4 (maximum speed reward). Then, it normalizes the reward value to a number in $[0, 1]$	27
5.6	CFG defining our DSL; a_i and o_i refer to one of the actions and observations of the game and n to a real number	28
5.7	Example program we provided to the participants for the Highway domain	32
5.8	Results of applying POLIS on the collected programs in our user study.	36
5.9	Comparing BUS* and Probe* for doing the synthesis task in POLIS on Lunar Lander (left) and Highway (right) game. Each point is a program a participant evaluated successfully using our system. These plots indicate that BUS* and Probe* perform better in Lunar Lander and Highway, respectively.	37

5.10	Example of a program one of the participants of our study wrote for the Highway domain.	41
5.11	POLIS's improved program for the program written by a participant of our study, which is shown in Figure 5.10	42
A.1	Describing the dynamics of the game, Highway.	52
A.2	Describing our DSL.	53
A.3	Playing Highway for at most 10 minutes before start coding. At each step of the game, the system shows observations at that step and time remaining. The Help button refers to a page that contains game and language description details.	54
A.4	Writing a program to play Highway with maximum reward. Participants can evaluate their programs on the game and see the observation at each step and the action their programmatic policies take, along with a timer that shows how much time remains. The Help button refers to a page that contains game and language description details. Participants can pause whenever they want and analyze the current observation and action. Also, they can stop the evaluation process at any time, providing an opportunity for a quick debug.	55
A.5	Survey and demographics questions.	56
B.1	Original POLIS's improved program for the program written by a participant of our study, which is shown in the main paper.	57

List of Algorithms

1	POLIS	18
2	Synthesizer - BUS*	20
3	Synthesizer - Probe*	22

Chapter 1

Introduction

Powerful artificial intelligence assistants for computer programmers have been developed thanks to recent advancements in large language models and program synthesis. These tools were designed to help both professional programmers and beginners in an online setting during programming tasks. For example, if a programmer is unsure how to approach a specific problem, a programming assistant like Copilot [1] can provide an initial solution to the programmer. Copilot can also speed up coding by auto-completing what programmers write, helping both beginner and expert programmers [2] complete the programming tasks.

While programmers usually communicate with these tools during programming, in this thesis, we propose a programming assistant that interacts with the programmer only *after* a compilable version of the program is available. Furthermore, this assistant attempts to modify a compilable program to optimize its behavior with respect to a measurable and comparable objective function.

As an intelligent programming assistant, we propose Program Optimization with Locally Improving Search (POLIS) for improving existing programs. POLIS considers each line of the program as an independent program synthesis task where the existing synthesizers are able to generate high-quality short programs. POLIS uses a novel local search algorithm on programs that iterates over lines of the program and performs a bottom-up search [3, 4] on each line’s synthesis task.

In the program-line space, POLIS can be considered a hill-climbing algorithm because it always chooses the best solution found by the bottom-up search for each line. Any measurable objective function that can be computed efficiently can be optimized using POLIS since the function needs to be evaluated for candidate programs during the synthesis.

To perform the synthesis tasks, we consider both the simple bottom-up search algorithm and a variation of the bottom-up search algorithm that uses a simple probabilistic context-free grammar to guide the search. Both algorithms do not need pre-training and have a few parameters to learn. However, other assistants, such as Copilot, rely on huge trained models with billions of parameters [5].

In this setting, we assume that programmers can implement compilable programs to solve a defined problem, knowing that their programs may not be optimal solutions nor optimal implementations. While Copilot and other existing assistant tools can help both novice and expert programmers in a general setting, POLIS may not be able to help beginner programmers who can not write compilable and syntax-free programs.

In addition to the program written by programmers, we assume there is a defined, measurable and comparable objective function for the optimization task, which might not be available in some domains. Such a function for complex problems can be very hard for humans to define or for computers to learn automatically. For example, in the reinforcement learning (RL) [6] community, inverse reinforcement learning (IRL) [7] approaches were proposed to infer reward functions automatically, usually by observing an expert’s behavior. However, IRL is still a hard and challenging problem [8]. For example, according to Finn *et al.*[9], different reward functions derived from an expert’s demonstration of the task may show the same behavior, but only one is the right reward function.

Though the assistant presented in this thesis, POLIS, is less general compared to the other approaches, it can be applied to problems in which humans know high-level

algorithmic solutions but are unsure about implementation details. While simple program synthesis algorithms can efficiently synthesize short programs to optimize an objective function in a reasonable short time, our system leverages this (often overlooked) feature of synthesizers to optimize the low-level details of human-written solutions.

To evaluate POLIS, we asked 27 programmers to write programs for playing two single-agent games, Lunar Lander and Highway, commonly used to evaluate reinforcement learning algorithms. POLIS was able to improve the score of *all* programs written by the participants, often by a large margin, while the simple synthesizer used in POLIS was unable to generate good programs from scratch for playing these games.

The modified programs are mostly similar to the original ones. This observation indicates that while humans are great at designing high-level program structures, they often fail to optimize important implementation details. In those cases, POLIS can be a programmer assistant to handle the suboptimality caused by human programmers.

The two main contributions of this thesis are as follows:

- Defining a novel problem setting for intelligent programming assistants where the assistant attempts to improve existing programs with respect to an objective function.
- Introducing POLIS, a system that employs a novel local search algorithm that leverages the ability of simple synthesizers to generate effective short programs.

We begin with Chapter 2, which discusses related work briefly. In Chapter 3, we provide background information on the bottom-up program synthesis variations we employ. The focus of Chapter 4 is on the proposed framework and the methodologies used. This is followed by Chapter 5, which delves into the application of POLIS in reinforcement learning, the user study, and the associated experimental results. Finally, we conclude the thesis in Chapter 6 and discuss future work.

Chapter 2

Related Work

In this chapter, we discuss related work to POLIS and the differences between POLIS and related work.

2.1 Program Synthesis

Program synthesis is a long-standing problem. The goal of program synthesis is to synthesize a program that satisfies a specification [10–13]. This problem has received much attention recently [3, 4, 14–21].

Syntax-guided synthesis (SYGUS) [22] is a general framework for finding a program that satisfies both semantic and syntactic specifications. The SYGUS has many plausible advantages compared to the general program synthesis framework [22]. For example, in the SYGUS framework, the size of the search space is reduced by the syntactic specification that comes in the form of context-free grammar (i.e., it limits the number of potential programs). Therefore, the synthesizer aims to find a program in the limited space consistent with the input-output examples (i.e., semantic specification) [20, 22].

Genetic programming [23] is a method for automatically generating a solution to a problem given a high-level description of the problem. In genetic programming, a population of computer programs is improved iteratively in a way comparable to natural selection in order to find an optimal solution to a problem.

POLIS is a framework that modifies an existing program by breaking it to small independent program synthesis problems and attempts to solve them while optimizing for an objective function.

2.2 Program Enhancement

Program refactoring and automated program repair are the two approaches that aim to enhance software systems. These approaches result in increasing the readability and maintainability of software systems and correcting the programmer’s mistakes automatically.

The process of modifying the internal structure of a program in order to improve its quality without affecting its external behavior is referred to as refactoring [24, 25]. By contrast, POLIS optimizes a program while possibly changing its external behavior.

Automated program repair (APR) refers to the process of fault localization in software and the development of patches using search-based software engineering and logic rules [26–28]. According to Goues *et al.*[27], repair patch approaches can be divided into two main approaches based on the patch type and the search strategy: heuristic-based repair and semantic-based repair [26].

The heuristic-based APR creates a large population of repair candidates for a given patch and chooses the one that passes all tests [26]. For example, Le Goues *et al.*[29] use genetic programming to develop bug-fixing patches without affecting software functionality.

The semantic-based technique (also known as the constraint-based approach) first extracts repair constraints and then uses program synthesis to generate a patch that satisfies those constraints [28].

Machine learning (ML) can sometimes improve the fault localization process and patch development [27]. Leveraging ML in APR has led to the new approach named learning-based repair that uses ML to build new patches based on previously created

patches and problem fixes, assuming that similar bugs will require similar solutions according to their features [26]. For example, Tufano *et al.*[30] train a NN model with a large dataset of bug fixes extracted from GitHub to predict the fixed version of a program.

Similar to APR methods, POLIS can also fix unintended programmer mistakes using program synthesis algorithms. However, unlike APR approaches, POLIS can change parts of the program that are not necessarily bugs but that contribute to suboptimal execution performance. To modify the program, POLIS makes use of inductive programme synthesis.

2.3 Intelligent Programming Assistant

Intelligent assistance for programmers is a widely explored area. For example, Ferdowsifard *et al.*[31] introduced a programming paradigm that allows the programmer to use a system for synthesizing instructions by defining input-output examples in the context of live programming. Blue-Pencil is another kind of assistant tool that does not require task specifications (i.e., input-output examples); thus, it identifies repetitive tasks that arise in programming and suggests transformations for such tasks [32]. Given a code transformation, reCode identifies other places of the code that would require similar changes [33]. For the code completion, Raychev *et al.*[34] introduced a statistical model and Guo *et al.*[35] introduced a model that leaves “holes” where the model is uncertain about the possible suggestions.

POLIS differs from these works in *how* it assists the programmer. Instead of real-time interactions during the program development, we consider the scenario where the programmer provides a complete and compilable version of their program. This allows us to leverage human-defined code structure and improve it with simple synthesizers that can write short and effective programs.

Algorithmic debugging techniques were first introduced by Shapiro[36]. By comparing the anticipated program behaviour with actual computation, this type of de-

bugger enables the user to concentrate on the program semantics [36][37]. It starts by constructing a tree that reflects the computation’s logic, then interacts with an external debugger which is usually the programmer and check the validity of the result of some nodes to find the buggy node [36][37]. When the debugger reports the fragment of code associated with the buggy node to the programmer, the debugging task ends [36][37].

POLIS tries to improve a program to avoid unintended behaviour by providing suggestions for different parts of the program, but it does not need external oracle to validate the computational result of each part of the program as it has access to an objective function. This is in contrast with the algorithmic debugging technique that needs a programmer in order to provide some information about the program and also solve the bug.

2.4 Interpretable Reinforcement Learning

Encoding a policy using programs or decision trees is a way to make the policy more amenable to interpretability. Neurally Directed Program Search (NDPS) [38] tries to synthesize a program that closely imitates a neural oracle for reinforcement learning domains. Viper also employs imitation learning to train decision tree encoding policies [39]. In order to provide better search guidance for synthesis, Propel trains neural policies that are not “too different” from the synthesized program [40]. Sketch-SA uses imitation learning to synthesize a sketch of a policy; the policy is synthesized from the sketch by evaluating it directly in the environment [41]. Oracle-free programmatically interpretable reinforcement learning (π -PRL) [42] leverages a differentiable language and trains the model using policy gradient methods. After training such a model, the model is mapped to a program as a policy.

All of these approaches encode policies learned by neural networks into the programs or decision trees by generating them from scratch. However, POLIS does not synthesize a complete program as a policy from scratch. Instead, it is designed to

modify and optimize an existing program while keeping its structure unchanged.

Chapter 3

Background

POLIS is a programmer assistant that improves programs by considering each line of the program as an independent program synthesis task. In this chapter, we explain two simple program synthesis algorithms that can be integrated with POLIS to solve such program synthesis problems.

In this chapter, we introduce an example of program synthesis problems and then describe two variations of the bottom-up search for solving it. We also explore how these variants reach the solution for that problem.

3.1 Problem Definition

Defining a program synthesis problem starts with a set of input-output examples that is known as the semantic specifications of the task. The set of input-output examples is defined in the form of $T = \{(i_1, o_1), (i_2, o_2), \dots, (i_n, o_n)\}$. The task is to synthesize a program that satisfies the semantic specifications. A program is considered a solution to the synthesis task if it correctly maps each input i_j to its corresponding output o_j . For our example, we consider Equation (3.1) as the semantic specification of the program we want to synthesize. The pairs in T show that the program has three numerical inputs and outputs one numerical value.

$$T = \{((1, 3, 2), 2), ((2, 2, 2), 2), ((3, 78, 0), 3), ((2, 7, 13), 4), ((3, 13, 5), 3)\} \quad (3.1)$$

Defining a Domain Specific Language (DSL) allows us to shape the search space

of possible programs that a synthesizer must explore. A DSL is defined as a context-free grammar (V, Σ, R, S) . V is the set of non-terminals, Σ is the set of terminals, and R is the set of relations that define the grammar’s production rules. S is the grammar’s start symbol. Equation 3.2 shows an example of a DSL where $V = \{S\}$, $\Sigma = \{x_1, x_2, x_3\}$, R are the relations (e.g., $S \rightarrow x_1$), and S is the start symbol. Σ is the input of the program in this DSL.

$$S \rightarrow \sqrt{S} \mid S + S \mid S - S \mid x_1 \mid x_2 \mid x_3 \quad (3.2)$$

This DSL allows programs with a sequence of operations like addition, subtraction and square root on the input data.

The DSL in Equation (3.2) and input-output examples in Equation (3.1) define our program synthesis problem. We then, in Section 3.2 and Section 3.3, describe how to find the solution to this problem using Bottom-Up Search and Probabilistic Guided-Bottom-Up Search.

3.2 Bottom-Up Search

Bottom-Up Search (BUS) is a dynamic-programming-based synthesis algorithm, first introduced in [3, 4]. BUS works by keeping a bank of all synthesized programs and builds new programs upon those previously generated programs. BUS starts with a bank consisting of only terminals in the grammar. Then in each iteration, it generates new programs by applying grammar rules to the programs retrieved from the bank. Finally, it adds the new programs to the bank [3, 4, 20]. After generating a new program based on a program retrieved from the bank, BUS evaluates it using the input-output examples and returns the first program that meets all the input-output examples as the solution to the synthesis problem. So whenever BUS finds the solution, the algorithm ends right away.

BUS explores the programs based on their costs. The cost of a program can be defined based on different features of the program; for example, the cost can be

defined as the size of the program [20, 43, 44]. The size of a program can be defined in different ways. However, a common metric to define the size of a program, which we use in our implementation, is the number of nodes in the Abstract Syntax Tree (AST) representing the programs. For example, the cost of terminals in the grammar is 1 and the cost of $x_1 + x_2 + x_3$ is 5. BUS starts with the terminals in the grammar, then BUS synthesizes all programs whose AST has two nodes, and so on until a solution is found. The solution that BUS finds is provably the smallest program that can solve the problem because BUS explores all possible programs.

Cost	# Programs	Bank
1	3	$\{x_1, x_2, x_3\}$
2	3	$\{\sqrt{x_1}, \sqrt{x_2}, \sqrt{x_3}\}$
3	75	$\{\sqrt{\sqrt{x_1}}, \sqrt{\sqrt{x_2}}, \sqrt{\sqrt{x_3}}, x_1 + x_1, \dots, x_1 - x_1, x_1 - x_2, \dots\}$
4	147	$\{\sqrt{\sqrt{\sqrt{x_1}}}, \dots, \sqrt{x_1 + x_1}, \dots, \sqrt{x_1 - x_1}, \dots, \sqrt{x_1 + x_1}, \dots, \sqrt{x_1 - x_1}, \dots\}$
5	12K	$\{\dots\}$
6	70K	$\{\dots\}$
7	...	$\{\dots, \sqrt{\sqrt{x_1 + x_2 + x_3}}, \dots\}$

Table 3.1: Programs generated for Equation (3.1) from the grammar in Equation (3.2) in the order of size. The size of a program is defined as the number of AST nodes of the program. To simplify the computation, we do not consider any pruning strategies to limit the search.

BUS, as a dynamic programming-based synthesis algorithm, at each iteration, synthesizes complete programs and reuses them to generate new ones in the next iterations. For example, given the simple DSL shown in Equation (3.2), Table 3.1 shows how BUS synthesizes programs to find the solution for the program synthesis problem at Equation (3.1). In Table 3.1, each row demonstrates the new programs of the bank in BUS and the number of programs synthesized at a certain cost. Table 3.1 shows that in the first iteration ($cost = 1$), BUS generates this set of programs $\{x_1, x_2, x_3\}$.

Then, in the next iteration ($cost = 2$), it generates the new programs based on the programs from the previous iteration $\{\sqrt{x_1}, \sqrt{x_2}, \sqrt{x_3}\}$, and so on. BUS iterates until the solution to the problem is found. In our example, BUS finds the solution at $cost = 7$, $p = \sqrt{\sqrt{x_1 + x_2} + x_3}$.

However, even for this very small DSL, the space of possible programs grows exponentially with the size of the programs. This growth has a direct impact on the runtime of the algorithm. As shown in Table 3.1, the synthesizer must generate more than 82K programs to find the solution. There are different techniques as pruning strategies for BUS. For example, as a simple method for pruning the search space, considering that all the programs generated by BUS are executable, Albarghouthi *et al.*[3] and Udupa *et al.*[4] suggested eliminating one of the two programs that are observationally equivalent during the synthesis. Two programs are observationally equivalent if they produce the same outputs given the same inputs. However, observationally equivalent and even state-of-the-art pruning strategies are not enough to prevent the BUS’s search space from growing exponentially [20].

One way to reduce BUS’s running time and prevent it from growing rapidly is to define the program’s cost efficiently so that BUS can find the solution program sooner in the search. Barke *et al.*[20] present an algorithm named Probe that guides BUS by learning a probabilistic model to guide the search.

3.3 Probabilistic Guided-Bottom-Up Search (Probe)

Probe is a guided bottom-up search algorithm for synthesizing programs according to a DSL. Probe redefines the cost function for the BUS algorithm. The main idea behind Probe is to explore programs in the order of decreasing likelihood instead of increasing size [20]. Probe learns a probabilistic context-free grammar (PCFG) to define the likelihood of a program as a guiding paradigm for BUS.

A PCFG defines a probability value for each production rule of the grammar. Probe starts by assigning a uniform distribution to the PCFG and updates the probability

of each rule during the search. Previous works [45, 46] observed that often there are considerable syntactic similarities between partial solutions (i.e., those that partially fulfill the semantic specification) and the final solution. For updating the PCFG, Probe takes advantage of this observation and updates the grammar based on the generated partial solutions during the search [20]. During the search, Probe encounters these partial solutions and modifies the PCFG by increasing the probability of the syntactic elements in the partial solutions [20].

Probe learns a PCFG by leveraging the set of input-output examples $T = \{(i_1, o_1), (i_2, o_2), \dots, (i_n, o_n)\}$. During search, given a set of programs P that Probe has synthesized so far, the probability of a production rule r , denoted $pr(r)$, is updated according to the following equation $pr(r) = z \cdot pr_u(r)^{1-\phi(P,T,r)}$. Here, $pr_u(r)$ is the probability of r according to the uniform distribution. Function $\phi(P, T, r)$ returns the largest proportion of input-output examples (i_j, o_j) that are satisfied by any program in P that uses rule r . Finally, in order to have a valid probability distribution, z is used as a normalizing factor. Intuitively, $pr(r)$ will receive a large probability value if a program that uses r correctly maps most of the inputs to their corresponding outputs. A large value of $pr(r)$ means the programs that use r should have a lower cost in search compared to the other programs in order to be explored sooner in the search. Therefore, Probe defines the cost of a program p as the negative sum of the log probabilities of each grammar rule in that program as in Equation (3.3).

$$cost(p) = \sum_{r \in p} [-\log(pr(r))] \quad (3.3)$$

Since Probe introduces a paradigm to guide BUS in order to explore the space of programs more efficiently, it synthesizes programs in the order of increased likelihood of a program being the solution. Therefore, Probe may find the solution sooner than the BUS as Probe uses a PCFG to guide the search. Starting with a uniform distribution for the PCFG, Probe learns the probability distribution for the grammar

rules using synthesized programs during the synthesis process. For example, in the DSL shown in Equation (3.2), Probe starts with $pr_u(S \rightarrow x_1) = \frac{1}{6} = 0.16$ for each rule. Then it eventually learns the probability values 0.28, 0.28, 0.02, 0.14, 0.14, 0.14 as a valid probability distribution for $\sqrt{S}, S+S, S-S, x_1, x_2, x_3$, respectively. Probe’s cost function definition is motivated by the idea that programs derived from production rules of the PCFG with higher probability values will have a lower cost and will be considered earlier in the search. In our example, the program $\sqrt{x_1 + x_2}$ is obtained from the initial symbol S by applying production rules $S \rightarrow \sqrt{S}, S \rightarrow S+S, S \rightarrow x_1$, and $S \rightarrow x_2$. According to the probability distribution above, the cost of the program is $\lfloor -\log(0.28) \rfloor + \lfloor -\log(0.28) \rfloor + \lfloor -\log(0.14) \rfloor + \lfloor -\log(0.14) \rfloor = 6$.

Table 3.2 shows the state of Probe’s bank at each iteration to find the solution for Equation (3.1) using Equation (3.2) with the cost of the rules 1, 1, 5, 2, 2, 2, for $\sqrt{S}, S+S, S-S, x_1, x_2, x_3$, respectively. Probe starts with the terminals in the grammar. The cost of each terminal based on the PCFG is 2. At each iteration, Probe builds new programs based on the programs in the bank and the grammar rules based on their costs. It finally finds the solution at $cost = 10$, which is $p = \sqrt{\sqrt{x_1 + x_2} + x_3}$. The most significant difference between Probe and the original BUS is the number of explored programs. To find the solution using BUS (Table 3.1), the synthesizer must explore more than 82K programs, and Probe (Table 3.2) only generates around 3K.

On the one hand, since POLIS considers each line as an independent synthesis problem, learning a PCFG for each problem may be beneficial to speed up the search. On the other hand, if the grammar and the synthesis problem are generally simple enough, Probe may not be efficient because it needs to learn the PCFG’s values and restart the search over and over, which may take more time than the original BUS. In our experiments, we use the original BUS and Probe to perform the synthesis tasks in POLIS.

Cost	# Programs	Bank
2	3	$\{x_1, x_2, x_3\}$
3	3	$\{\sqrt{x_1}, \sqrt{x_2}, \sqrt{x_3}\}$
4	3	$\{\sqrt{\sqrt{x_1}}, \sqrt{\sqrt{x_2}}, \sqrt{\sqrt{x_3}}\}$
5	12	$\{\sqrt{\sqrt{\sqrt{x_1}}}, \dots, x_1 + x_1, x_1 + x_2, x_1 + x_3, \dots\}$
6	48	$\{\sqrt{\sqrt{\sqrt{\sqrt{x_1}}}}, \dots, \sqrt{x_1 + x_2}, \dots, x_1 + \sqrt{x_1}, \dots, \sqrt{x_1} + x_1\}$
7	93	$\{\dots\}$
8	354	$\{\dots\}$
9	3200	$\{\dots\}$
10	...	$\{\dots, \sqrt{\sqrt{x_1 + x_2 + x_3}}, \dots\}$

Table 3.2: Programs generated for Equation (3.1) from the probabilistic context-free grammar in Equation (3.2) in order of cost defined by Probe. To simplify the computation, we do not consider any pruning strategies to limit the search.

Chapter 4

Program Optimization with Locally Improving Search (POLIS)

In this chapter, we define a program as the input of POLIS. Then, we describe how to optimize a program with POLIS.

4.1 Program Definition

A program is a string of production rules of a grammar that used to define a DSL. It is a deterministic function written/synthesized that outputs the program's behavior given the input. Let D be a DSL and $\llbracket D \rrbracket$ be the (possibly infinite) set of programs that can be written based on D . Each program $p \in \llbracket D \rrbracket$ is defined by a pair $\{T, L\}$, where T is a multi-set of terminal symbols, and L defines a partition of symbols from T into the program lines. L defines how a programmer organizes the symbols in T in a text editor. Note that two programs that have identical functionality could have different partitions L .

4.2 POLIS: a Programming Assistant

POLIS is a programming assistant that takes an objective function F that maps a program to a real value and a program $p \in \llbracket D \rrbracket$ as inputs. It outputs a program $p' \in \llbracket D \rrbracket$ that is at least as good as p and approximates a solution for $\arg \max_{p \in \llbracket D \rrbracket} F(p)$.

4.2.1 Local Search

POLIS is local search that uses bottom-up synthesis algorithms to improve the behavior of an existing program. The key idea behind POLIS is to consider each line of the program as an independent program synthesis task. It tries to synthesize a better program for each line in the context of the whole program without changing the other lines and their orders. A better program for a given line of code results in a better evaluation score compared to the original program of that line. Algorithm 1 shows the body of the local search algorithm employed by POLIS. Algorithm 1 can be considered a type of hill-climbing algorithm that traverses the optimization landscape defined by a program and a set of input-output pairs and a DSL to optimize the program.

The algorithm takes an existing program p and two-time limits, t and t_l , as inputs. t limits the overall running time of the search, and t_l is the maximum running time allowed to spend optimizing each line of code. The search returns a new program, p' , that is at least as good as p .

While there is time available to improve the input program, the search iterates through each line (the for loop in line 3), and it attempts to synthesize a program that replaces the code in the i -th line of p such that the objective function F is improved. This is achieved with a call to the synthesizer (line 4). The synthesizer can call any program synthesis algorithm that returns a better program that improves the objective function F . In our experiment, we evaluate both BUS and Probe [20] with some modifications as the synthesizer.

4.2.2 BUS for Optimization Problems

The original BUS algorithm uses the inductive program synthesis framework, which takes the synthesis problem and input-output examples. Then it returns a program that meets the input-output examples. In order to use BUS for an optimization problem with respect to an objective function, we modify the original algorithm

Algorithm 1: POLIS

Data: Initial program p , overall time limit t , time limit per line t_l ,
input-output examples ε , \mathcal{G}
Result: Improved program p'

```
1 while Not-Timeout( $t$ ) do
2    $p' \leftarrow p$ 
3   for  $i \leftarrow 1$  to Number-of-Lines( $p$ ) do
4      $p \leftarrow \text{Synthesizer}(p, i, t_l, \varepsilon, \mathcal{G})$ 
5     if  $F(p) = F(p')$  then
6       return  $p'$ 
```

to evaluate the programs using an objective function as well as the input-output examples to guide the search. We refer to the new algorithm as BUS*. POLIS calls BUS* in line 4 of Algorithm 1 as the synthesizer.

Unlike BUS, BUS* does not return a program that meets all the input-output examples. Instead, it returns the program with the best evaluation score among others with respect to the objective function. In this section, we highlight the difference between BUS and BUS*.

Algorithm 2 shows BUS* algorithm and the highlighted lines indicate the difference between BUS and BUS* algorithms. BUS* receives the program p , the line number i indicating the synthesizer should synthesize a program for it, and the grammar \mathcal{G} defining a DSL along with the input-output examples ε and parameter k . BUS* also takes a value t_l as an input which limits the running time of the algorithm. It starts by initializing the bank with an empty set and $best_l$ with the initial program of the i -th line (line 1 to 3 of Algorithm 2). The synthesizer attempts to replace the i -th line with another line that optimizes the objective function. When the synthesizer's search time is up, it delivers a version of the original program p in which the i -th line has been replaced with the best program it found during the synthesis process. If the i -th line of the program already produces the best F -value, or if the synthesizer runs out of time before discovering a better line, the synthesizer may return the program unaltered.

To replace the i -th line, BUS^* starts with $c = 1$. Then after generating all the programs with $\text{cost} = c$, c is incremented by 1 for the next iteration and so on (line 14).

Since the metric F , which we attempt to optimize, may be computationally expensive, BUS^* leverages multi-level evaluation methods. BUS^* uses the set of input-output examples to reduce the computational cost of evaluating programs as we explain next. Instead of computing the metric F for all programs generated in search, we keep a list of the current k -best programs with respect to an output-agreement metric. The output-agreement metric accounts for the number of inputs each program correctly maps to the output paired with each input in T . The output-agreement metric we use is computed as $\frac{\sum_{o \in T} \mathbb{1}[p(i_j)=o_j]}{|T|}$, where T is the set of input-output examples, $\mathbb{1}[\cdot]$ is the indicator function, $p(i_j)$ and o_j are the output returned by the program p and example output in T , respectively, for input i_j . At line 8, the output-agreement score of the new program is computed based on the formula above. BUS^* evaluates the metric F only for the programs in the k -best set (line 11). Therefore, the number of evaluating F reduces significantly.

Once BUS^* runs out of time, it returns the best program in the set of k best with respect to the F score, not with respect to the output-agreement metric.

4.2.3 Probe for Optimization Problems

Since the Probe algorithm is based on the BUS algorithm [20], we develop Probe^* based on BUS^* shown in Algorithm 3. The input and output of Probe^* are the same as BUS^* ; however, \mathcal{G} in Probe^* is a probabilistic context-free grammar. Probe^* is a program synthesis algorithm so POLIS calls it in line 4 of Algorithm 1 as the synthesizer.

Probe^* starts with initializing the bank B and the update list $UList$ that is used to update the grammar \mathcal{G} , with empty lists and parameter j with k at lines 1 and 2. It initializes other parameters as BUS^* (lines 3 and 4). Although the original Probe algorithm begins with a uniform \mathcal{G} , Probe^* updates the \mathcal{G} with the initial program

Algorithm 2: Synthesizer - BUS*

Data: Program p , line i , time limit t_l , input output examples ε , grammar \mathcal{G} , parameter k

Result: Optimized program p_{opt}

```
/* Initialize the state of bank B */
1  $B \leftarrow \{\}$ 
2  $l_{init} \leftarrow \text{GetLine}(i)$ 
3  $best\_l \leftarrow l_{init}$ 
4 for  $c$  from 1 to 100 do
    /* Iterate over all programs whose cost =  $c$  and are not observationally
       equivalent to another program in  $B$  */
5   for  $l$  in  $\text{New-Programs}(B, c, \mathcal{G})$  do
6     if  $\text{Not-Timeout}(t_l)$  then
7        $p_{new} \leftarrow \text{ModifyLine}(p, i, l)$ 
8        $score \leftarrow [\langle in, \llbracket p_{new} \rrbracket(in) \rangle | \langle in, o \rangle \in \varepsilon] \cap \varepsilon$ 
9       if  $score$  is in among the best  $k$  programs'  $score$  then
10        if  $\text{Eval}(p, l, i) > \text{Eval}(p, best\_l, i)$  then
11           $best\_l \leftarrow l$ 
12      else
13         $p_{opt} \leftarrow \text{ModifyLine}(p, i, best\_l)$ 
14      return  $p_{opt}$ 
15  $p_{opt} \leftarrow \text{ModifyLine}(p, i, best\_l)$ 
```

of the i -th line before beginning synthesis (line 5). This inductive bias directs the synthesizer to initially explore programs structurally similar to the one it was given. This enables us to include human insight in the synthesis.

While there is time for synthesis, Probe* synthesizes programs with $cost = c$ at each iteration, given \mathcal{G} and previously generated programs in the bank B . It then computes its score based on output-agreement method at line 11. If the program is among the best k best programs based on their scores, Probe* compares the evaluation of l and $best_l$ in the context of p and updates the best program for i -th line $best_l$ (line 20). Probe* also adds l to the updated list $UList$ if l is one of the best k programs. Like BUS*, Probe* also leverages the set of input-output examples to reduce the computational cost of evaluating programs using output-agreement metrics and evaluates a program l in the game environment only if it is among k best programs.

We use the programs in the set of k best programs to update the probability distribution of the PCFG Probe* learning. In our implementation at line 18 at Algorithm 3, Probe* first updates its PCFG once it adds k programs to the set of best k programs. The second update is performed when $j = k - 1$ new programs are added to the set since the first update; the third update happens after adding $j = k - 2$ programs since the second update, and so on. The value of j stops decrementing once it reaches 1. At the beginning of the search, when Probe* quickly finds programs with better output-agreement values, the PCFG is updated less often. However, later in the search, when it is hard to find programs with better output-agreement values than those already explored in the search, Probe* updates the PCFG as soon as a program enters the set of k best programs ($j = 1$). Like the original Probe algorithm, the search is restarted once the PCFG \mathcal{G} is updated. Probe* finally returns the modified programs with the optimized i -th line.

Once Probe* runs out of time, the i -th line of p is replaced with a program with the best game score $best_l$.

Algorithm 3: Synthesizer - Probe*

Data: Program p , line i , time limit t_l , input output examples ε , PCFG \mathcal{G} , parameter k

Result: Optimized program p_{opt}

```
/* Initialize the state of bank  $B$  and update list  $UList$  */
1  $B, UList \leftarrow \{\}, \{\}$ 
2  $j \leftarrow k$ 
3  $l_{init} \leftarrow \text{GetLine}(i)$ 
4  $best\_l \leftarrow l_{init}$ 
5  $\mathcal{G} \leftarrow \text{Update}(\{l_{init}\}, \varepsilon)$ 
6 while Not-Timeout( $t_l$ ) do
7   for  $c$  from 1 to 100 do
8     /* Iterate over all programs whose cost =  $c$  and are not
9       observationally equivalent to another program in  $B$  */
10    for  $l$  in New-Programs( $B, c, \mathcal{G}$ ) do
11      if Not-Timeout( $t_l$ ) then
12         $p_{new} \leftarrow \text{ModifyLine}(p, i, l)$ 
13         $score \leftarrow [\langle in, \llbracket p_{new} \rrbracket(in) \rangle | \langle in, o \rangle \in \varepsilon] \cap \varepsilon$ 
14        if  $score$  is in among the best  $k$  programs'  $score$  then
15          if  $\text{Eval}(p, l, i) > \text{Eval}(p, best\_l, i)$  then
16             $best\_l \leftarrow l$ 
17             $UList = UList \cup \{l\}$ 
18            if number of programs in  $UList = j$  then
19               $\mathcal{G} \leftarrow \text{Update}(UList, \varepsilon)$ 
20               $j \leftarrow \max(j - 1, 1)$ 
21               $UList \leftarrow \{\}$ 
22              break
23          else
24             $p_{opt} \leftarrow \text{ModifyLine}(p, i, best\_l)$ 
25            return  $p_{opt}$ 
26  $p_{opt} \leftarrow \text{ModifyLine}(p, i, best\_l)$ 
```

Chapter 5

Experiments

In this chapter, we describe the games used as problem domains in our user study, the experimental details of POLIS in the context of writing programs for playing such games, and the experimental design used in our study.

5.1 Problem Domains

We evaluate POLIS on two games in our study: Lunar Lander and Highway (Figure 5.1). These games were chosen to reduce possible biases participants could have toward existing solutions to the problems. Solutions to traditional programming tasks might be easily found online, or participants could already know the solution to the problems. In contrast, these two games are used as benchmark problems for reinforcement learning algorithms and not as programming assignments. Additionally, both domains have a clear metric: the game score.

5.1.1 Lunar Lander

In the Lunar Lander game, a player needs to control three (Boolean) thrusters of a spaceship trying to land on the moon. Eight features describe the Lunar Lander state at each environment step, as shown in Figure 5.2. At each step, the spacecraft (lander) can fire its left, right, or main (bottom) thrusters, or it can take no action. After every step of the game, the player gets a reward and the total reward of an

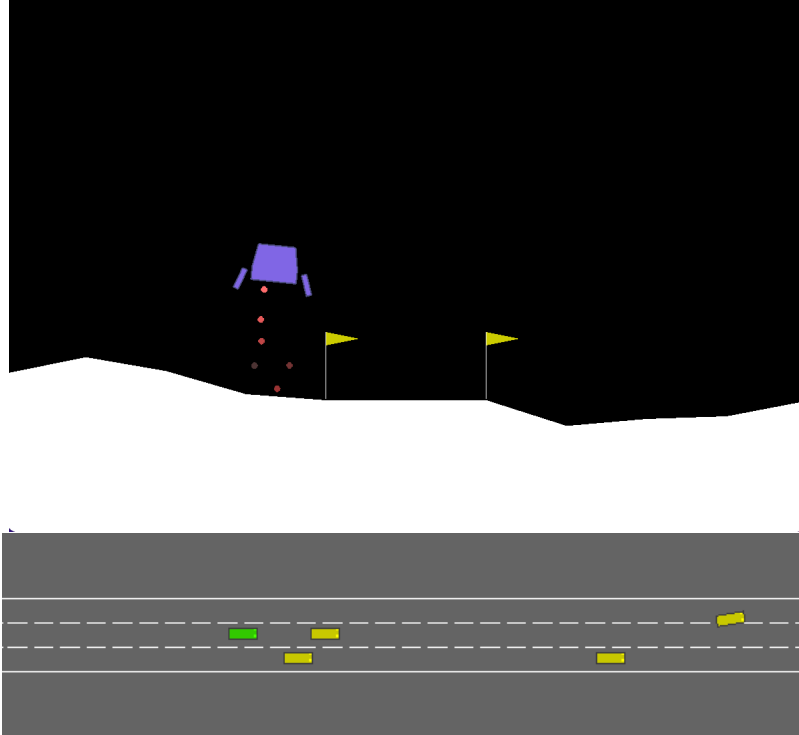


Figure 5.1: Lunar lander (top) and Highway (bottom)

episode is the sum of the rewards for all steps in that episode. The game score is maximized if the player does not use the thrusters unnecessarily and gently reaches the landing pad. Figure 5.3 demonstrate how to calculate game score at each step. The game is considered solved if the player can collect 200 points or more. We use the LunarLander-v2 implementation from Open AI Gym [47].

5.1.2 Highway

In the Highway game, a player controls a green car on a three-lane highway. The game score is higher when the player drives fast, avoids collisions, and spends more time in the right lane. Each car is represented by four features, as shown in Figure 5.5. The player can change lanes or increase or reduce speed. In this version of the Highway environment, the agent can see its car features as well as the other two nearest cars, so the total features describing the state for the agent add up to 12 (see Equation (5.5)). However, the number of cars presented on the highway may be more than three at a

$$state = \begin{cases} 0: \text{horizontal coordinate of the lander,} \\ 1: \text{vertical coordinate of the lander,} \\ 2: \text{horizontal velocity,} \\ 3: \text{vertical velocity,} \\ 4: \text{orientation of the lander in the space,} \\ 5: \text{angular velocity,} \\ 6: \text{if left leg touches the ground 1, otherwise 0,} \\ 7: \text{if right leg touches the ground 1, otherwise 0} \end{cases} \quad (5.1)$$

$$action = \begin{cases} 0: \text{do nothing,} \\ 1: \text{fire left orientation engine,} \\ 2: \text{fire main engine,} \\ 3: \text{fire right orientation engine} \end{cases} \quad (5.2)$$

Figure 5.2: In the Lunar Lander game, the state of the spaceship is determined based on eight features at each step as in Equation (5.1). There are four actions available for the player as shown in Equation (5.2)

$$reward = \begin{cases} \text{is increased/decreased the closer/further the lander is to the landing pad,} \\ \text{increased/decreased the slower/faster the lander is moving,} \\ \text{decreased the more the lander is tilted (angle not horizontal),} \\ \text{increased by 10 points for each leg that touches the ground,} \\ \text{decreased by 0.03 points if a side engine is firing,} \\ \text{decreased by 0.3 points if the main engine is firing} \end{cases} \quad (5.3)$$

Figure 5.3: According to the Open AI Gym [47] documentation, the reward after each step of the game is calculated based on these rules. The player at the end of each episode receives an additional reward of -100 or +100 points for crashing or landing safely respectively.

$$car = \begin{cases} \text{horizontal coordinate of the car,} \\ \text{vertical coordinate of the car,} \\ \text{horizontal velocity,} \\ \text{vertical velocity} \end{cases} \quad (5.4)$$

$$state = \begin{cases} [0:3]: \text{ player's car,} \\ [4:7]: \text{ the first nearest car,} \\ [8:11]: \text{ the second nearest car} \end{cases} \quad (5.5)$$

$$action = \begin{cases} 0: \text{ go to the left lane,} \\ 1: \text{ do nothing,} \\ 2: \text{ go to the right lane,} \\ 3: \text{ go faster,} \\ 4: \text{ go slower} \end{cases} \quad (5.6)$$

Figure 5.4: In the Highway game, each car is represented by four features as seen in Equation (5.4). At each step of the game, the player can see the green car's and the two nearest car's features as shown in Equation (5.5). Equation (5.6) indicates five actions available at each step at Highway.

time. After every step, the player gets a reward and the total reward of an episode is the sum of the rewards for all steps withing that episode. Figure 5.7 indicates the reward function in Highway. If there is a collision between player's car and other cars, the episode terminates and the player gets -1 for the last step. The maximum reward for Highway is 50. We use the implementation of Leurent[48].

5.2 Experimental Details

To optimize programmatic policies written for the two games mentioned above with POLIS, we first explain the Domain Specific Language used for writing programmatic policies for those games in Section 5.2.1. Then, we describe how to collect input-output examples in such a problem in Section 5.2.2 and Section 5.2.3. Finally, we describe how to deal with local optimum solutions found by POLIS in Section 5.2.4.

$$reward = \begin{cases} -1: & \text{if the car collides with a vehicle,} \\ \frac{(v-15)}{(30-15)} * 0.4 + 0.6 & \text{(for driving on the right most lane)} \end{cases} \quad (5.7)$$

Figure 5.5: At each step of Highway, the reward is calculated as Equation (5.7) where v is the speed of the car and $[15, 30]$ is the speed range. For v less than 15, the speed reward is 0 and for v more than 30, the reward speed is 0.4 (maximum speed reward). Then, it normalizes the reward value to a number in $[0, 1]$

5.2.1 POLIS Domain Specific Language

A general-purpose language like Python defines a very large program space. However, in this work, we consider a domain-specific language (DSL) to define a much smaller space of programs for solving a given programming task. We define a python-like programming language as our DSL to make it simple for humans to write and understand programs since many programmers know Python. Our DSL allows an arbitrary number of (un)nested if-then-else structures and arbitrarily large chains of commands. As shown in Figure 5.6, our DSL consists of helper functions (e.g., addition, subtraction, \dots), action primitives specific for each environment, variables, and control flows (i.e., if-else statements and Boolean/logical operators).

Since our DSL is based on Python programming language, we define the structure of a program as a list of independent statements and each statement considered as a line in that program. A program based on our DSL can contain (un)nested if-else and assignment statements. Moreover, POLIS tries to improve the conditions of if-else statements and the expression statements of assignment statements.

5.2.2 Input-Output Examples

BUS* and Probe* use input-output examples for evaluating each program during the synthesis. Probe* also learns a PCFG for each program synthesis task using input-output examples. In the regular program synthesis tasks, these would be given by a set of test cases. Previous approaches [38–40] suggested using deep neural networks (DNN) as an oracle to guide the synthesis of policies encoded as computer

$$\begin{aligned}
P &::= \mathbf{def} \text{ heuristic}(o): S \mathbf{return} \text{ action} \\
S &::= SS \mid \mathbf{if}(C): S \mathbf{else}: S \mid \mathbf{if}(C): S \mathbf{elif}(C): S \mathbf{else}: S \\
&\quad \mid V_{def} \mid A_{assign} \\
C &::= C B C \mid E \\
E &::= o_i \mid V_{name} \mid n \mid E M E \mid \text{pow}(E, E) \mid \text{sqrt}(E) \\
&\quad \mid \text{log}(E) \mid -(E) \mid \text{abs}(E) \\
V_{def} &::= V_{name} = E \\
A_{assign} &::= \text{action} = a_i \\
B &::= \text{and} \mid \text{or} \mid < \mid <= \mid != \mid == \mid >= \mid > \\
M &::= + \mid - \mid * \mid /
\end{aligned}$$

Figure 5.6: CFG defining our DSL; a_i and o_i refer to one of the actions and observations of the game and n to a real number

programs. They employed imitation learning [49] to generate programs for playing games. We use the approach Verma *et al.*[49] introduced to learning programmatic policies. Specifically, they train a neural policy that generates a set of input-output pairs: for a set of observations o (input), we store the neural policy’s chosen action a (output). We use DQN [50] to train a neural policy π for 2000 episodes. We then let the agent follow π in the environment for 2000 steps and collect all the observation-action pairs along with their Q -values.

5.2.3 Highlights

We further optimize the computational cost of our evaluation function by using a small number of input-output examples. Instead of collecting a large number of observation-action pairs uniformly at random, we collect a small set of observations based on the Highlights ranking of importance [51]. Highlights ranks a set of observations according to the most considerable difference in Q -values for different actions available at a given observation. Highlights calculates the importance of an observation-action pair based on Equation (5.8).

$$I(s) = \max_a Q_{(s,a)}^\pi - \min_a Q_{(s,a)}^\pi \quad (5.8)$$

We run policy π in each game and collect the top 400 observation-action samples according to their I -values in our experiments, chosen from a set of 2000 observation-action samples. The number of observation-action samples has a direct impact on the running time of POLIS. We tried 1000 and 400 input-output examples and the bigger number increased the running time of the algorithm significantly. Therefore, we selected 400 Highlight observation-action examples for our experiments.

5.2.4 Restarts

POLIS’s hill-climbing algorithm traverses the optimization landscape defined by the initial program and the set of input-output pairs. POLIS’s greedy approach to optimization could lead the algorithm to become stuck in locally optimal solutions. An effective strategy for dealing with local optimum solutions is to restart the search from a different starting location in the optimization landscape once the search stops in a local optimum [52]. Every time we restart the search, we train a different DQN agent to generate a new set of input-output examples. This enables us to restart the search while allowing for a potentially different initial starting location within the optimization landscape.

5.2.5 Bayesian Optimization

The real numbers n in our DSL (see Figure 5.6) are set using Bayesian optimization [53]. The BUS* and Probe* enumeration procedure generate programs with the symbol n , then run the Bayesian optimizer to replace them with real values. The optimizer chooses the real values to replace n while attempting to optimize for the output-agreement metric. In Algorithms 2 and 3, after generating new programs, their numerical values are optimized by calling BayesOpt function. BayesOpt optimizes the newly synthesized program l for line i of program p using input-output examples ε .

5.3 User Study Design

We conducted a user study to collect different programs that optimize the game score of Lunar Lander and Highway, which we introduced in Section 5.1. In this section, we describe our user study in detail. We developed a web-based system to conduct our user study based on HippoGym [54] and advertised the study on mailing lists of graduate and undergraduate Computing Science students at University of Alberta. The ethics ID number for this study is Pro00113586.

In the beginning, we had all participants digitally sign a consent form. In the form, we indicate that they would be asked to write a program for playing a computer game. We also explained that the amount of money they would get would be proportional to their final program’s game score; higher game scores would result in higher monetary values. In other words, the more points their program got in the game, the more money they would get. The minimum compensation was 15 CAD. We used the following formulae to compute the compensation of each participant: $15 + (100 + x) \times (1/30)$ and $15 + x \times (1/5)$ for Lunar Lander and Highway, respectively. x is the participants’ average game score over 100 and 25 episodes of Lunar Lander and Highway, respectively (an episode is completed when the player finishes landing the spaceship in Lunar Lander and when the player crashes the car or a time limit is reached in the Highway). The participants received a maximum of 25 CAD, even if the formula results in a value >25 .

After agreeing to the terms of the study and signing the consent form, each participant was given a random assignment to one of the two games. Then, they read a tutorial on the game that had been assigned to them. In the tutorial, we explained the features in each observation passed as the program’s input parameters and the actions available to the player. In addition, our tutorial had a few examples with screenshots of the game showing situations where different actions were applied to different game observations. The final step of the tutorial was a multiple-choice ques-

tion about the game; immediate feedback was provided to the participant showing whether they chose the correct or wrong answer. If an answer was incorrect, the participant would have as many attempts as needed to answer it correctly.

Following the game tutorial, each participant read a tutorial about our DSL. The tutorial presented the CFG shown in Figure 5.6 and explained the Boolean and algebraic functions and the programming structures supported by our DSL. Similarly to the game tutorial, we provided several examples of programs that can be written in our DSL. The tutorial finished with a multiple-choice question where the participant had to select, among four options, the program that was accepted in our DSL; the participant had as many attempts as needed to answer the question correctly.

Before writing a program for playing the game, the participant had the opportunity to play the game using their keyboard for 10 minutes. The observation values and the game score each participant obtained for each game run were displayed in real-time on our graphical user interface. The participant had the option to stop playing at any moment (within the 10 minutes allowed by our system) and start writing their program. Our goal with this step of the study was to provide the participant with the opportunity to develop a strategy for playing the game, which they could then attempt to encode into their programs.

We provided the participants with a Python-like editor, where the keywords of the DSL are highlighted. The editor also had an example of a simple program for playing the game. For example, for Highway, we provided the program shown in Figure 5.7. This program moves the car to the right lane if the car is not already there; the car does nothing otherwise.

Additionally, the game they were writing code for was shown on our interface so the participants could run their programs and observe how those programs perform in the game. Similarly to the interface used when the participant played the game, we displayed the observation values and the game score the program obtained in real-time. If the participant wanted to inspect the values of the observations, they could

```
1 def heuristic(o):
2     action = 0
3     if o[1] < 8:
4         action = 2
5     else:
6         action = 1
7     return action
```

Figure 5.7: Example program we provided to the participants for the Highway domain

pause the simulation at any time. Additionally, our interface allowed the participants to go back to the tutorials while playing the game or writing their program.

We made sure to store all compilable and syntactically correct programs the participants evaluated so that we could use them as input for our evaluation. The experiment duration, which included both playing the game and writing a program, was 60 minutes, maximum 10 minutes for playing the game and 50 minutes for writing a program. Within the 50-minute time limit, the participant could submit the final version of their program at any time. We calculated the participant’s monetary compensation based on their final program submission. The participant then answered demographic questions before finishing the experiment.

5.4 Results

In this section, we report the demographic data of the participants in our user study. We abbreviate standard deviation as SD and interquartile range as IQR to describe the demographic information. We then report the computational results of applying POLIS on participants’ programs. The results emphasize the significance of POLIS’s performance on optimizing those programs. Finally, we conclude this chapter by analyzing an example program before and after applying POLIS on it to find out how POLIS optimize such a program.

5.4.1 Demographics

In our user study, 40 people consented to participate, and 26 completed the survey. The average age was 20.96 (SD of 4.13), with ages ranging from 18 to 40; 20 participants identified themselves as male, 5 as female, and 1 withheld gender information. Most (20) had received or were pursuing undergraduate education, 4 had completed high school, and 2 were pursuing post-secondary training. Most participants (25) had not done any form of game artificial intelligence (AI) research, and about half had not taken any AI courses. More than one-third of the participants (10) rarely or never played computer games, and others occasionally or often played computer games.

We asked about the participants’ programming experience: 22 had more than one year of experience, and 4 had less than a year. We also asked about their knowledge of Python, how hard it was to write a program in our DSL, and how hard it was to write a program for solving the game. We used a 5-point, Likert-like scale: 1 being “novice” in Python and “very easy” for writing programs, to 5 being “expert” in Python and “very hard” for writing programs. The median response to these three questions were: 3 (IQR = 1), 2.5 (IQR = 2), and 4 (IQR = 1), respectively. On average, the participants had some experience in Python and found it easy to use our DSL, but found it hard to write a program to play the game. To evaluate POLIS, we considered the data from those who submitted at least one working program (different from the example program we provided), resulting in 27 participants (one of them did not complete the survey).

5.4.2 Computational Results

During the 50 minute time limit for writing programs, participants wrote different programs and submitted what they thought to be the best one. We stored every program the participants evaluated in this process. Tables 5.1 and 5.2 show how participants performed in our study. For each participant, we report the number of programs they wrote and evaluated, the average of their programs’ scores and the

ID	Attempts Count	Avg. Score	SD	Min	Max	Last	Time
1	29	-169.63	121.08	-545.41	-93.15	-106.07	00:25:02
2	37	-167.34	173.41	-557.24	44.33	44.33	00:47:24
3	29	-252.20	282.12	-1022.52	-68.21	-68.21	00:42:57
4	2	-125.12	0.00	-125.12	-125.12	-125.12	00:06:50
5	3	-125.12	0.00	-125.12	-125.12	-125.12	00:02:30
6	14	-349.34	226.60	-813.57	-118.65	-118.65	00:46:50
7	14	-159.05	91.20	-430.96	-110.72	-110.72	00:39:33
8	35	-174.53	132.33	-544.51	-32.60	-473.64	00:47:15
9	73	-163.85	150.07	-797.12	-34.21	-89.09	00:50:02
10	13	-79.67	26.60	-156.53	-45.10	-45.10	00:10:20
11	42	-249.98	324.51	-1260.72	11.45	11.45	00:48:47
12	48	-32.20	56.81	-160.53	52.84	43.73	00:48:07

Table 5.1: A summary of Participant’s performance for Lunar Lander

standard deviation, minimum and maximum scores, as well as their last program’s score. The total time they spent writing programs is shown in both tables as time in the format of hh:mm:ss.

As shown on Table 5.1 and 5.2, participants evaluated more programs for Lunar Lander compared to Highway (see column “Attempts Count”). This may suggests that participants had to see the performance of their programs even after small changes for Lunar Lander, while for Highway, once they understood the logic of the game, they were able to write good programs without having to evaluate them too often. One reason could be that controlling a car in a highway is more intuitive than controlling a lander in the space. Interestingly, 9 participants did not submit their best programs (“Max” is larger than “Last”): see participants 1, 2, 5, 8, 9, 10, 11, 12, and 15 in Table 5.2. By contrast, only 4 participants did not submit their best programs in Lunar Lander.

We ran POLIS 10 times and use 5 restarts in each run. We report the average game

ID	Attempts Count	Avg. Score	SD	Min	Max	Last	Time
1	8	6.24	2.09	5.06	11.23	5.40	00:47:34
2	6	7.05	1.87	6.21	11.23	6.21	00:14:14
3	12	16.03	5.77	5.70	24.28	24.28	00:46:47
4	5	7.25	0.00	7.25	7.25	7.25	00:10:01
5	3	6.81	0.78	6.21	7.91	6.31	00:01:28
6	12	14.66	7.38	7.87	25.14	25.05	00:24:16
7	21	12.68	7.31	7.47	30.25	30.25	00:48:04
8	14	11.78	2.14	9.42	17.21	11.23	00:47:00
9	5	11.39	0.73	10.51	12.74	11.23	00:11:51
10	13	14.84	7.19	8.90	31.19	11.55	00:42:26
11	7	11.77	6.92	5.06	27.39	7.55	00:15:51
12	11	12.95	6.60	6.83	30.25	6.83	00:44:19
13	8	23.04	8.59	8.37	28.98	28.98	00:30:09
14	9	18.09	8.68	8.86	31.95	31.95	00:31:41
15	10	23.47	7.62	8.08	35.71	12.44	00:11:02

Table 5.2: A summary of Participant’s performance for Highway

score of a POLIS’s improved program as the average of 10 runs. Figure 5.8 summarizes the POLIS results for Lunar Lander (left) and Highway (right). Each point (x, y) in the plots represents a program a participant evaluated successfully in our study. The x -value of each point is the average game score a program obtained in 100 and 25 episodes of Lunar Lander and Highway respectively. The y -value of each point is the average game score of a program that POLIS improves obtained in 100 and 25 episodes of the games. We consider both BUS* and Probe* to the synthesis task in POLIS for both games. Thus, the blue points represent POLIS’s result using Probe* and green points represent POLIS’s result using BUS*.

All points in both plots are above the diagonal $y = x$ line, which means that POLIS was able to improve the performance of all programs that participants wrote, for

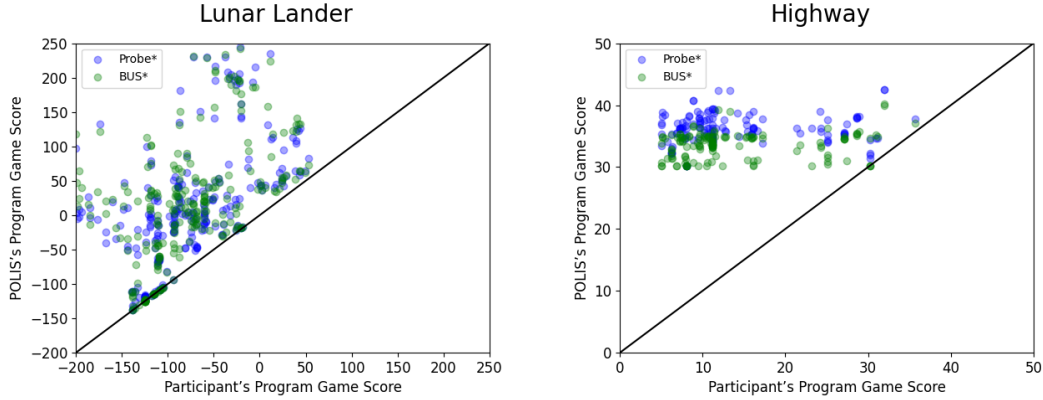


Figure 5.8: Results of applying POLIS on the collected programs in our user study.

both domains, regardless of which synthesis algorithm was used. In Lunar Lander we observe a trend in the data points where better human-written programs allowed for better POLIS-improved programs.

To compare Probe* and BUS* better, Figure 5.9 summarizes the POLIS results with BUS* and Probe* for Lunar Lander (left) and Highway (right). Each point (x, y) in the plots represents a program a participant evaluated successfully in our study. We apply both BUS* and Probe* to optimize each program. The x -value of each point is the average game score the optimized version of the program using BUS* obtained in 100 and 25 episodes of Lunar Lander and Highway, respectively. The y -value of each point is the average game score of the optimized version of the program that POLIS improves by applying Probe*.

Most of the points in the left plot in Figure 5.9 (Lunar Lander) are under the diagonal $y = x$ line, which means that POLIS, in combination with BUS*, was able to improve the performance of most of the participants' programs more than Probe*. However, this is the other way around for the right plot in Figure 5.9 (Highway), where almost all the points are above the diagonal $y = x$ line. This observation suggests that POLIS with Probe* improved the performance of almost all programs considerably more than BUS*. Thus, in our experiment, for Lunar Lander, BUS*, and for Highway, Probe* does the synthesis tasks.

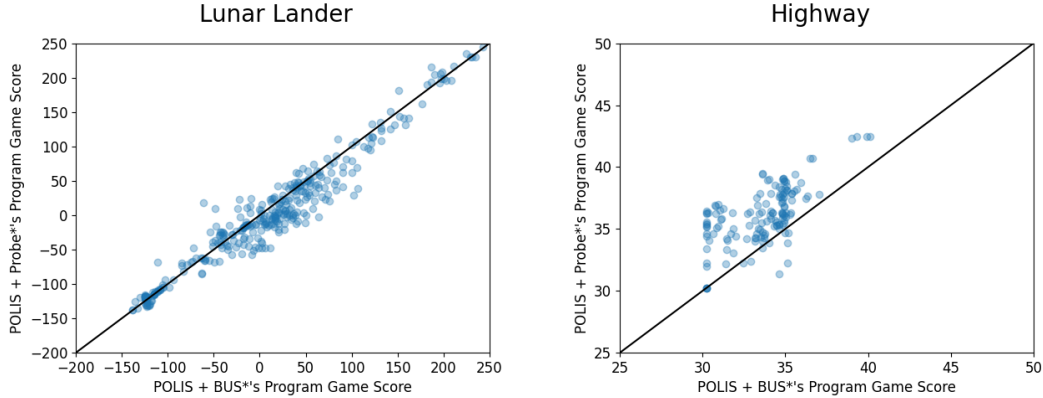


Figure 5.9: Comparing BUS* and Probe* for doing the synthesis task in POLIS on Lunar Lander (left) and Highway (right) game. Each point is a program a participant evaluated successfully using our system. These plots indicate that BUS* and Probe* perform better in Lunar Lander and Highway, respectively.

POLIS’s average score is higher for all programs written in our study. A Wilcoxon signed-rank test indicated that the difference between the original programs’ performance and the average of the improved programs’ performance was significant and also pointed to large effect size for the average results of both domains: 0.624 for Lunar Lander ($p < 4.9 \times 10^{-4}$) and 0.621 for Highway ($p < 3.1 \times 10^{-5}$).

Tables 5.3 and 5.4 show the results for Lunar Lander and Highway, respectively. Here, each participant is represented by an ID which is the same as Tables 5.1 and 5.2. The game score of both the participants’ and POLIS’s programs is an average of the score of the program obtained in 100 of Lunar Lander in Table 5.3 and 25 episodes of Highway in Table 5.4. The game score shown for POLIS is the average over 10 independent runs of the system. Each run of POLIS can result in different game scores due to the random initialization of the DQN’s neural network policy π used to generate input-output pairs. We also present the standard deviation and minimum and maximum game scores across these 10 independent runs. We applied POLIS on all programs written by participants and for each participant selected the best improved program p' , based on the program’s average score and its original program p . The original program score in both tables refers to the score of p . The program the participants submit as their final programs are not necessarily the programs with

ID	Original Program		POLIS Program					
	Score	LoC	Score	SD	Min	Max	Edited LoC	Time
1	-449.72	8	-22.17	9.18	-35.03	-7.95	11.40	01:36:43
2	-196.33	14	68.85	37.46	-22.09	105.62	13.90	03:13:38
3	-125.12	9	-123.96	3.45	-125.12	-113.61	1.10	00:17:25
4	-125.12	14	-122.68	7.30	-125.12	-100.79	1.10	00:28:14
5	-118.65	24	101.11	49.32	-4.07	173.94	27.50	09:49:21
6	-110.72	29	0.44	35.71	-34.20	90.06	9.80	04:44:42
7	-89.09	24	82.63	39.95	7.91	165.62	23.40	06:21:02
8	-87.42	16	56.19	21.45	23.15	96.78	17.70	03:56:36
9	-70.13	10	65.05	17.06	38.68	101.32	5.90	01:37:13
10	-21.50	52	252.90	6.60	240.77	260.86	30.50	13:50:26
11	40.26	24	143.57	16.33	112.17	164.82	20.90	08:16:07
12	52.84	13	74.45	7.99	63.84	89.30	5.30	01:55:21

Table 5.3: Game score improvements for Lunar Lander

the highest score compared to their previously submitted programs, as shown in Tables 5.1 and 5.2. The number of lines of code (LoC) indicates how many lines the original program has. In both tables, we sort the rows according to the participant’s program game score, from lowest (top) to highest (bottom). The number of edited lines (Edited LoC) refers to the average number of lines that POLIS modifies during optimization. The total time of the process for a program, including collecting the input-output examples and optimizing the program, is shown in both tables as time in the format of hh:mm:ss. We also present the average number of car collisions in Highway (Hits) in Table 5.4.

For Lunar Lander, POLIS provided significant improvements to most of the participants’ scores (e.g., IDs 2 and 10), but there are a few participants that POLIS was not able to improve their scores notably. For example, considering that we run

ID	Original Program			POLIS Program						
	Score	LoC	Hits	Score	SD	Min	Max	Edited LoC	Hits	Time
1	5.09	40	25	38.88	1.25	35.38	40.27	12.40	0.80	13:57:40
2	6.21	6	25	35.45	1.17	33.16	38.12	6.90	2.30	03:37:54
3	6.97	12	25	37.35	0.87	35.82	38.32	10.20	2.50	08:55:50
4	7.25	24	25	36.73	1.63	34.25	40.03	7.60	3.10	13:10:12
5	7.91	8	25	35.21	1.77	31.11	36.95	7.00	2.70	03:56:36
6	9.61	12	25	38.24	2.15	34.64	41.19	9.40	1.90	06:34:02
7	10.80	6	24	37.71	1.20	35.31	39.45	7.60	2.70	04:23:16
8	11.23	20	24	39.38	0.77	37.99	40.42	16.40	0.60	13:11:27
9	11.23	10	24	38.03	0.68	37.15	39.02	9.50	1.20	06:53:34
10	11.55	14	21	36.69	0.83	35.36	38.16	10.00	2.40	08:54:34
11	14.23	16	23	38.80	0.53	37.79	39.45	8.80	1.20	06:54:39
12	15.33	9	22	36.80	0.00	36.80	36.80	7.00	4.00	04:41:14
13	24.23	12	14	39.20	0.83	37.79	40.61	5.50	1.90	05:44:09
14	31.95	20	2	42.50	0.06	42.40	42.55	7.00	0.00	11:35:32
15	35.71	13	2	37.89	0.24	37.53	38.16	3.40	1.20	06:29:17

Table 5.4: Game score improvements for Highway

POLIS ten times on each program, for IDs 3 and 4, POLIS was unable to improve their scores at some runs. In addition, the total time and the number of edited lines for the programs of participants 3 and 4 are much smaller than for the other programs. This indicates that POLIS quickly stopped at a local minimum for these programs.

It is interesting to note that regardless of how well the participant’s programs performed in Highway, POLIS enhanced the performance of all programs so that they could achieve a game score between 30 to 45 (Table 5.4 and Figure 5.8). We hypothesize that the participants identified the program’s structure necessary to play Highway well. This makes the programs for that game more receptive to POLIS’s enhancements because POLIS does not alter the program’s general structure and only

makes local improvements. A possible explanation for the discrepancy between the Lunar Lander and Highway outcomes is that the latter problem (how to drive a car) is more intuitive than the former (how to land a spaceship). The Lunar Lander results point to a limitation of POLIS, which is its inability to improve programs that need simultaneous changes to several parts of the code. Finally, POLIS substantially reduced the number of car collisions in the Highway domain, in some cases, from more than 20 to approximately 1 collision.

5.4.3 Representative Program

The program shown in Figure 5.10 is representative of a program written by one of the participants of our study for the Highway domain based on our DSL 3.2. As shown in Figure 5.6, our DSL is a subset of Python Language, and it ensures that all programs define a function called “heuristic” that receives the player’s observation of the game o and returns an action to be performed. We refer to the program in Figure 5.10 as p in this section. This program obtains an average game score of 6.8 over 25 episodes. Figure 5.11 shows POLIS’s improved program for p , which we will refer to as p' . We lightly edited p' for readability. The original p' can be found in Appendix B. POLIS’s p' obtains an average game score of 39.0 over 25 episodes, a major improvement over the original program. The participant who wrote p made a mistake while writing the first “if” statement of p in line 3. In line 3 of p , there are two Boolean conditions connected with an “or” operation. For example, the left Boolean condition checks whether $o[5]$ is equal to $o[1]$ and if $o[5] - o[1] > 200$. However, the two parts of the expression cannot be simultaneously true, as once $o[5]$ is equal to $o[1]$, we have that $o[5] - o[1]$ is zero and can not be > 200 . The same conclusion can be drawn for the Boolean expression on the right side of “or.” As a result, the player never slows down (action 4). While the participant’s intention with this “if” statement was likely to slow the car down if the player’s car was in the same lane as the nearest car on the road (condition $o[5]$ is equal to $o[1]$ returns true if the cars are

```

1 def heuristic(o):
2     action = 0
3     if (o[5] == o[1] and o[5]-o[1] > 200) or (o[9] == o[1] and o
4         [9]-o[1] > 200):
5         action = 4
6     elif (o[5] == o[1] and o[5]-o[1] <= 200) or (o[9] == o[1]
7         and o[9]-o[1] <= 200):
8         if o[1] == 4:
9             if o[9] < 4:
10                action = 2
11            else:
12                action = 0
13        else:
14            action = 3
15    return action

```

Figure 5.10: Example of a program one of the participants of our study wrote for the Highway domain.

on the same lane).

POLIS not only fixed the problem with the Boolean condition in the participant’s program but also changed the player’s strategy. Instead of slowing down if another car is in the same lane, p' only slows down when changing lanes; $o[3]$ is the car’s velocity on the y -axis, which is not zero when the car is changing lanes. Since the car is changing lanes, $o[1]$ cannot be zero, as $o[1]$ equals zero, constantly representing the car being on the leftmost lane. In contrast with p , p' changes lanes when another car is in the same lane. This strategy is encoded in the “elif” structure of the program, which can be translated as: if the nearest car is in the same lane ($o[5]$ is equal to $o[1]$), then move to the right lane (action 2), if the car is not already in the rightmost lane (lines 7 and 8); move to the left lane if already in the rightmost lane.

POLIS’s improved program p' prefers to drive on the rightmost lane if the car driving on the same lane is not the nearest one (i.e., there is still time to change lanes). The program maximizes its score by driving in the rightmost lane. Finally,

```
1 def heuristic(o):
2     action = 0
3     if o[1] and o[3]:
4         action = 4
5     elif o[5] == o[1] or o[9] == o[1]:
6         if o[1] == o[5]:
7             if o[1] < 7.9317:
8                 action = 2
9             else:
10                action = 0
11        else:
12            action = 2
13    else:
14        action = 1
15    return action
```

Figure 5.11: POLIS’s improved program for the program written by a participant of our study, which is shown in Figure 5.10

POLIS’s program does nothing (action 1) if it is not changing lanes and there is no car in front of it. POLIS’s strategy is a cautious one as the car slows down as it changes lanes, but it never speeds up. This cautious strategy achieves a much higher game score than the participant’s program.

Hence, we argue that the person who wrote p can easily understand p' and its strategy. Additionally, we hypothesize that the POLIS’s optimized program p' is understandable for someone who at least knows the input and output of the “heuristic” function. Thus, debugging and verifying POLIS’s programs and extending them to add more details to their strategies are straightforward and simple for humans.

Chapter 6

Conclusion and Future Work

In this thesis, we presented a system, Program Optimization with Locally Improving Search (POLIS), that is able to improve existing programs with respect to a measurable, real-valued metric. POLIS leverages an often overlooked strength of simple synthesizers: the ability to generate short and effective programs. POLIS divides the problem of improving an existing implementation into smaller subproblems by considering each line of the program as an independent program synthesis task. This way, POLIS employs two variations of bottom-up search synthesizer: the original bottom-up search implementation and Probe that learns a probabilistic context-free grammar for each subproblem. The synthesizer attempts to replace a single line of the original program at a given time while all the other lines remain unchanged. We conducted a user study with 27 participants who wrote programs for playing two games. POLIS was able to improve the performance of the programs of all participants, often by a large margin. Since POLIS performs local changes with bottom-up search synthesizers, its modified program shares the same structure as the original program. The similarity of the programs allowed us to understand how POLIS improved the performance of a representative program from our study. The results of our experiments suggest that POLIS can be used as an effective programming assistant in scenarios where one is interested in improving an existing program with respect to a measurable, comparable metric.

As a direction for the future work, we can analyze the relations of (a) the number of input-output examples, and (b) evaluating any program during the search with function F , with POLIS’s performance and running time. Also, it would be interesting to compare Highlights and random approaches for selecting the observation-action pairs in games and explore their impact on the POLIS’s performance. Moreover, it would be interesting to see whether the participant’s background in programming has any impact on POLIS’s performance.

One of the limitations of POLIS is regarding the DSL introduced in Figure 5.6. For example, the control flow of our DSL only includes conditional statements (i.e., if-else statements). Moreover, our DSL supports some predefined mathematical functions and does not accept imported libraries and other helper functions. An interesting extension of POLIS could consider the setting that the DSL accepts iterative statements (i.e., for and while loops). The programmers could benefit from defining their helper functions and a large library of predefined functions by DSL. Therefore, it would be interesting to discover whether POLIS can optimize the programs written in such a more general-purpose DSL.

Another limitation of our work is that our experimental domains are two single-agent games: Lunar Lander and Highway, used to evaluate reinforcement learning algorithms. Using POLIS in commercial software and programming tasks could be an essential next step as it shows the general usage of POLIS.

An important feature of POLIS’s optimized programs is that they are human-understandable, as highlighted by the discussion in Section 5.4.3. Compared to the original programs written by our participants, POLIS’s modified program retains the majority of the original program’s structure. This observation draws attention to an important advantage of POLIS. As an interesting direction for future research is to do a user study to see whether programmers can understand POLIS’s optimized programs. If the programmer who wrote the original code is able to understand POLIS’s improved version and can learn from its solution, POLIS can be used as a

teaching tool to highlight students' mistakes and also provide a working solution for each mistake. This tool will help students analyze their mistakes and reform them as an opportunity to learn more about programming and problem-solving skills.

Bibliography

- [1] N. Friedman, *Introducing GitHub Copilot: your AI pair programmer*, <https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/>, [Online; accessed 11-August-2022], 2021.
- [2] S. Barke, M. B. James, and N. Polikarpova, “Grounded copilot: How programmers interact with code-generating models,” *CoRR*, vol. abs/2206.15000, 2022. DOI: 10.48550/arXiv.2206.15000. arXiv: 2206.15000. [Online]. Available: <https://doi.org/10.48550/arXiv.2206.15000>.
- [3] A. Albarghouthi, S. Gulwani, and Z. Kincaid, “Recursive program synthesis,” in *International Conference Computer Aided Verification, CAV*, 2013, pp. 934–950.
- [4] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, “Transit: Specifying protocols with concolic snippets,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, 2013, pp. 287–296.
- [5] M. Chen *et al.*, *Evaluating large language models trained on code*, 2021. DOI: 10.48550/ARXIV.2107.03374. [Online]. Available: <https://arxiv.org/abs/2107.03374>.
- [6] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018, ISBN: 0262039249.
- [7] A. Y. Ng and S. J. Russell, “Algorithms for inverse reinforcement learning,” in *Proceedings of the Seventeenth International Conference on Machine Learning*, ser. ICML ’00, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, 663–670, ISBN: 1558607072.
- [8] S. Arora and P. Doshi, “A survey of inverse reinforcement learning: Challenges, methods and progress,” *Artificial Intelligence*, vol. 297, p. 103 500, 2021, ISSN: 0004-3702. DOI: <https://doi.org/10.1016/j.artint.2021.103500>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0004370221000515>.
- [9] C. Finn, S. Levine, and P. Abbeel, “Guided cost learning: Deep inverse optimal control via policy optimization,” in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML’16, New York, NY, USA: JMLR.org, 2016, 49–58.

- [10] R. J. Waldinger and R. C. T. Lee, “Prow: A step toward automatic program writing,” in *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, ser. IJCAI’69, Washington, DC: Morgan Kaufmann Publishers Inc., 1969, 241–252.
- [11] D. C. Smith, “Pygmalion: A creative programming environment,” Stanford University, Tech. Rep., 1976. [Online]. Available: <http://worrydream.com/refs/Smith%20-%20Pygmalion.pdf>.
- [12] M. Fraňová, “A methodology for automatic programming based on the constructive matching strategy,” in *EUROCAL ’85*, B. F. Caviness, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 568–569, ISBN: 978-3-540-39685-7.
- [13] Y. Kodratoff, M. Franova, and D. Partridge, “Logic programming and program synthesis,” in *Systems Integration ’90. Proceedings of the First International Conference on Systems Integration*, 1990, pp. 346–355. DOI: 10.1109/ICSI.1990.138700.
- [14] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, “Deepcoder: Learning to write programs,” in *Proceedings International Conference on Learning Representations (ICLR)*, Apr. 2017. [Online]. Available: <https://openreview.net/pdf?id=rkE3y85ee>.
- [15] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli, “Robustfill: Neural program learning under noisy I/O,” in *Proceedings of the 34th International Conference on Machine Learning*, D. Precup and Y. W. Teh, Eds., ser. Proceedings of Machine Learning Research, vol. 70, PMLR, 2017, pp. 990–998.
- [16] A. Kalyan, A. Mohta, O. Polozov, D. Batra, P. Jain, and S. Gulwani, “Neural-guided deductive search for real-time program synthesis from examples,” in *International Conference on Learning Representations*, 2018. [Online]. Available: <https://openreview.net/forum?id=rywDjg-RW>.
- [17] R. Shin *et al.*, “Synthetic datasets for neural program synthesis,” *ArXiv*, vol. abs/1912.12345, 2019.
- [18] K. Ellis *et al.*, “Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning,” *CoRR*, vol. abs/2006.08381, 2020. [Online]. Available: <https://arxiv.org/abs/2006.08381>.
- [19] W. Lee, K. Heo, R. Alur, and M. Naik, “Accelerating search-based program synthesis using learned probabilistic models,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Association for Computing Machinery, 2018, 436–449. DOI: 10.1145/3192366.3192410. [Online]. Available: <https://doi.org/10.1145/3192366.3192410>.
- [20] S. Barke, H. Peleg, and N. Polikarpova, “Just-in-time learning for bottom-up enumerative synthesis,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–29, Nov. 2020, ISSN: 2475-1421. DOI: 10.1145/3428295. [Online]. Available: <http://dx.doi.org/10.1145/3428295>.

- [21] R. Ji, Y. Sun, Y. Xiong, and Z. Hu, “Guiding dynamic programming via structural probability for accelerating programming by example,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, 2020. DOI: 10.1145/3428292. [Online]. Available: <https://doi.org/10.1145/3428292>.
- [22] R. Alur *et al.*, “Syntax-guided synthesis,” Oct. 2013, pp. 1–17. DOI: 10.1109/FMCAD.2013.6679385.
- [23] J. Koza and R. Poli, “Genetic programming,” in Jan. 2005, pp. 127–164, ISBN: 9780387283562. DOI: 10.1007/0-387-28356-0_5.
- [24] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
- [25] M. F. Aniche, E. Maziero, R. S. Durelli, and V. H. S. Durelli, “The effectiveness of supervised machine learning algorithms in predicting software refactoring,” *CoRR*, vol. abs/2001.03338, 2020. arXiv: 2001.03338. [Online]. Available: <https://arxiv.org/abs/2001.03338>.
- [26] T. Sharma, M. Kechagia, S. Georgiou, R. Tiwari, and F. Sarro, “A survey on machine learning techniques for source code analysis,” *CoRR*, vol. abs/2110.09610, 2021. arXiv: 2110.09610. [Online]. Available: <https://arxiv.org/abs/2110.09610>.
- [27] C. L. Goues, M. Pradel, and A. Roychoudhury, “Automated program repair,” *Commun. ACM*, vol. 62, no. 12, 56–65, Nov. 2019, ISSN: 0001-0782. DOI: 10.1145/3318162. [Online]. Available: <https://doi.org/10.1145/3318162>.
- [28] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, “Semfix: Program repair via semantic analysis,” in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 772–781. DOI: 10.1109/ICSE.2013.6606623.
- [29] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “Genprog: A generic method for automatic software repair,” *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.
- [30] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, “An empirical investigation into learning bug-fixing patches in the wild via neural machine translation,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018, Montpellier, France: Association for Computing Machinery, 2018, 832–837, ISBN: 9781450359375. DOI: 10.1145/3238147.3240732. [Online]. Available: <https://doi.org/10.1145/3238147.3240732>.
- [31] K. Ferdowsifard, A. Ordookhanians, H. Peleg, S. Lerner, and N. Polikarpova, “Small-step live programming by example,” in *Proceedings of the Annual ACM Symposium on User Interface Software and Technology*, New York, NY, USA: Association for Computing Machinery, 2020, 614–626, ISBN: 9781450375146. DOI: 10.1145/3379337.3415869. [Online]. Available: <https://doi.org/10.1145/3379337.3415869>.

- [32] A. Miltner *et al.*, “On the fly synthesis of edit suggestions,” *Proceedings of the ACM Programming Languages*, vol. 3, no. OOPSLA, 2019. DOI: 10.1145/3360569. [Online]. Available: <https://doi.org/10.1145/3360569>.
- [33] W. Ni, J. Sunshine, V. Le, S. Gulwani, and T. Barik, “ReCode: A Lightweight Find-and-Replace Interaction in the IDE for Transforming Code by Example,” in *The 34th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST ’21, Virtual Event, USA: Association for Computing Machinery, 2021, 258–269, ISBN: 9781450386357. DOI: 10.1145/3472749.3474748. [Online]. Available: <https://doi.org/10.1145/3472749.3474748>.
- [34] V. Raychev, M. Vechev, and E. Yahav, “Code completion with statistical language models,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA: Association for Computing Machinery, 2014, 419–428, ISBN: 9781450327848. DOI: 10.1145/2594291.2594321. [Online]. Available: <https://doi.org/10.1145/2594291.2594321>.
- [35] D. Guo, A. Svyatkovskiy, J. Yin, N. Duan, M. Brockschmidt, and M. Allamanis, “Learning to complete code with sketches,” in *International Conference on Learning Representations*, 2022. [Online]. Available: <https://openreview.net/forum?id=q79uMSC6ZBT>.
- [36] E. Y. Shapiro, “Algorithmic program debugging,” AAI8221751, Ph.D. dissertation, USA, 1982.
- [37] R. Caballero, A. Riesco, and J. Silva, “A survey of algorithmic debugging,” *ACM Comput. Surv.*, vol. 50, no. 4, 2017, ISSN: 0360-0300. DOI: 10.1145/3106740. [Online]. Available: <https://doi.org/10.1145/3106740>.
- [38] A. Verma, V. Murali, R. Singh, P. Kohli, and S. Chaudhuri, “Programmatically interpretable reinforcement learning,” *CoRR*, vol. abs/1804.02477, 2018. arXiv: 1804.02477. [Online]. Available: <http://arxiv.org/abs/1804.02477>.
- [39] O. Bastani, Y. Pu, and A. Solar-Lezama, “Verifiable reinforcement learning via policy extraction,” in *Proceedings of the International Conference on Neural Information Processing Systems*, Curran Associates Inc., 2018, 2499–2509.
- [40] A. Verma, H. M. Le, Y. Yue, and S. Chaudhuri, “Imitation-projected programmatic reinforcement learning,” in *Proceedings of the International Conference on Neural Information Processing Systems*, Curran Associates Inc., 2019.
- [41] L. C. Medeiros, D. S. Aleixo, and L. H. S. Lelis, “What can we learn even from the weakest? Learning sketches for programmatic strategies,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI Press, 2022.
- [42] W. Qiu and H. Zhu, “Programmatic reinforcement learning without oracles,” in *International Conference on Learning Representations*, 2022. [Online]. Available: <https://openreview.net/forum?id=6Tk2noBdvxt>.
- [43] R. Alur, A. Radhakrishna, and A. Udupa, “Scaling enumerative program synthesis via divide and conquer,” in *TACAS*, 2017.

- [44] M. Koukoutos, E. Kneuss, and V. Kuncak, “An Update on Deductive Synthesis and Repair in the Leon Tool,” in *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016*, R. Piskac and R. Dimitrova, Eds., ser. EPTCS, vol. 229, 2016, pp. 100–111. DOI: 10.4204/EPTCS.229.9. [Online]. Available: <https://doi.org/10.4204/EPTCS.229.9>.
- [45] H. Peleg and N. Polikarpova, “Perfect Is the Enemy of Good: Best-Effort Program Synthesis,” in *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, R. Hirschfeld and T. Pape, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 166, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 2:1–2:30, ISBN: 978-3-95977-154-2. DOI: 10.4230/LIPIcs.ECOOP.2020.2. [Online]. Available: <https://drops.dagstuhl.de/opus/volltexte/2020/13159>.
- [46] K. Shi, J. Steinhardt, and P. Liang, “Frangel: Component-based synthesis with control structures,” *CoRR*, vol. abs/1811.05175, 2018. arXiv: 1811.05175. [Online]. Available: <http://arxiv.org/abs/1811.05175>.
- [47] G. Brockman *et al.*, *OpenAI Gym*, 2016. DOI: 10.48550/ARXIV.1606.01540. [Online]. Available: <https://arxiv.org/abs/1606.01540>.
- [48] E. Leurent, *An environment for autonomous driving decision-making*, <https://github.com/eleurent/highway-env>, 2018.
- [49] A. Verma, V. Murali, R. Singh, P. Kohli, and S. Chaudhuri, “Programmatically interpretable reinforcement learning,” *CoRR*, vol. abs/1804.02477, 2018. arXiv: 1804.02477. [Online]. Available: <http://arxiv.org/abs/1804.02477>.
- [50] V. Mnih *et al.*, “Human-level control through deep reinforcement learning,” *nature*, vol. 518, no. 7540, pp. 529–533, 2015.
- [51] D. Amir and O. Amir, “Highlights: Summarizing agent behavior to people,” in *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, ser. AAMAS ’18, Stockholm, Sweden: International Foundation for Autonomous Agents and Multiagent Systems, 2018, 1168–1176.
- [52] H. Hoos and T. Stützle, *Stochastic Local Search: Foundations & Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004, ISBN: 1558608729.
- [53] J. Snoek, H. Larochelle, and R. P. Adams, “Practical Bayesian Optimization of Machine Learning Algorithms,” in *Proceedings of the International Conference on Neural Information Processing Systems*, Red Hook, NY, USA: Curran Associates Inc., 2012, 2951–2959.
- [54] M. E. Taylor, N. Nissen, Y. Wang, and N. Navidi, “Improving reinforcement learning with human assistance: An argument for human subject studies with HIPPO gym,” *CoRR*, vol. abs/2102.02639, 2021. arXiv: 2102.02639. [Online]. Available: <https://arxiv.org/abs/2102.02639>.

Appendix A: Graphical User Interface

Figures A.1 to A.5 show screenshots of the graphical user interface of our experiment.

Instructions:

In this study, you are going to play with Highway. Once you have learned how to play it well (meaning gain more points), you will be asked to implement your best strategy for playing this game, in the language we define later (which is a simplified version of python).

Please note that you will have only 10 minutes for understanding the game by playing it and then 50 minutes for writing your strategy.

Description of Game:

Highway is the game you will write a strategy for. You should write a function to control your car to drive on the right most lane with highest speed. In that case, you will receive maximum reward at each step. Maximum reward at each step is 1, and the total number of steps for each episode is 60. So in total, you can collect 60 points at each episode.

This Highway, has 3 lanes. The car starts randomly on a lane with a random horizontal velocity and 0 vertical velocity. The goal of this game is to drive on the right most lane with high horizontal speed (max speed is 30).

State vector, which is represented by 's' as the input of your strategy function, consists of 12 states: the horizontal and vertical coordinates, the horizontal and vertical velocities of your car and two nearest cars to yours, respectively.

There are five discrete actions available:

- 0: go to the left lane
- 1: do nothing
- 2: to go the right lane
- 3: increase speed (horizontal)
- 4: decrease speed (horizontal)

Here is an example of dynamics of Highway. After each image (state of the game) you see the observation values in a table. You may find useful information about the game by analysing these values. For example, you can find the range of the right most lane of the highway by comparing the y position values. Remember that you will be asked to implement your strategy for driving a car in this highway, using a very simple programming language and these values.

(1) The very first step: restart the game. You see the initial position of the green and yellow cars.



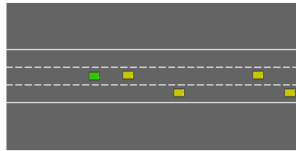
	x position	y position	speed at x	speed at y
green car (agent)	200.83293	0	25	0
1st nearest car	226.2009	4	21.72741	0
2nd nearest car	251.14046	8	21.120872	0

(2) Here you see the transition from state (1) (above) by taking action 2 (go to the right lane) by agent (green car). In this lane, green car will receive a partial reward of being in the right most lane.



	x position	y position	speed at x	speed at y
green car (agent)	225.60094	3.331477	24.851133	2.7241867
1st nearest car	245.40291	4	17.669235	0
2nd nearest car	269.72583	8	17.127172	0

(3) Here you see the transition from state (2) (above) by taking action 4 (decrease the speed) by agent (green car).



	x position	y position	speed at x	speed at y
green car (agent)	-248	-4.17	-20.8	-0.02
1st nearest car	-263	4	-18.38	0
2nd nearest car	-286	8	-16.29	0

(4) Here you see the transition from state (3) by taking action 2 (go to the right lane) by agent (green car). This lane is considered as the right most lane.



	x position	y position	speed at x	speed at y
green car (agent)	-268	-7.35	-20	-2.5
1st nearest car	-282	4	-19	0
2nd nearest car	-302	8	-14.7	0

Follow Up Question:

Based on your understanding of the Highway game, please answer the following question.

The green car is in lane 0 now. After taking action 0 and then action 2, what is the y value in the observation?

- impossible
- 8
- 4
- 0

Please click "Submit" when you're ready to move on.

Submit

Figure A.1: Describing the dynamics of the game, Highway.

Programming Language:

You must use the below *python-style* language to implement your strategy. But there are a few built-in functions that you can use in your program, and also, you can not import any libraries.

You are supposed to write your best strategy for playing the game in a "heuristic" function. We have developed an agent that receives your heuristic function, compiles it, and decides which action to take based on your strategy at each step of the game. The input of this function is the state vector of the game, which is described before, and the output is the action.

Please find the grammar of the language below.

```
Program P ::= def heuristic(s) : S return action
Statement S ::= S S
              | if(C) : S else : S
              | if(C) : S elif(C) : S else : S
              | Vdef
              | Aassign
Condition C ::= (C) opo (C)
              | E
Expression E ::= Oindex
              | Vname
              | N
              | (E) opo (E)
              | pow(E, E)
              | sqrt(E)
              | log(E)
              | -(E)
              | abs(E)
Variable Vdef ::= Vname = E
Action Aassign ::= action = actionindex
Number N ::= int
            | float
```

$op_o \in \{and, or, <, <=, ! =, ==, >=, >\}$.

$op_m \in \{+, -, *, /, \}$.

$action_{index} \in \{\text{available actions in the game}\}$.

$O_{index} \in \{\text{available observations in the game}\}$

Grammar of our python-style language

Description of the grammar:

For implementing your strategy, in each line, you can either define a [variable](#) and assign an expression to it or use [if/else statements](#).

NOTE: If you use "if" it must be followed by "else." But you can have "elif" as many as you want between "if" and "else". You must follow python rules for naming your variables. You must have only one function named "heuristic".

Please Do Not Use: lists, True, False, tuple and any other python features that are not in our language.

Here is a list of primitive functions/operations you can use in your implementation:

- `neg(n)` (it returns the $-n$)
- `abs(n)` (it returns $|n|$)
- `log(n)`
- `sqrt(n)`
- `pow(n, m)` (it returns n^m)
- $n + m$
- $n - m$
- $n * m$
- n / m

The following operations must be used just for condition of "if-else" statements:

- $n > m$
- $n \geq m$
- $n == m$
- $n != m$
- $n < m$
- $n \leq m$

n and m can be:

1. The inputs of the heuristic function, which are observations.
2. Any numbers.
3. Variables that you have defined earlier.
4. An expression which is a combination of (1), (2) and (3) and the primitive functions.

If you are using more than one "+" or "-" or "*" or "/" (any binary operations), please use parentheses "("..."") to separate them from each other. For instance: $a + (b + (c + d))$

Please look at the following examples:

- Variable definition examples:
 - `a = 2*3`
 - `b = log(a)`
 - `c = (a - b) * sqrt(a)`
- if/else statement example:
 - if $a > c$ and $pow(a, 2) < b + c$:
 - # here in the body of "if", you can define some variables or "if-else"s
 - else:
 - # here in the body of "else", you can define some variables or "if-else"s

Like the above example, you can have [inner/nested](#) or [sequential](#) "if-else" statements.

Feel free to leave any [comments](#) in your code using the same rules as python. We encourage you to describe your strategy step by step, inside your code by comments, only if you have time.

Follow Up Question:

Based on your understanding of the grammar of the language, please answer the following question.

Here are some examples of the program written by people. Please select the one that is grammatically correct.

```
def heuristic(s0, s1):
    var1 = [s0, s1]
    result = var1[0]
    return result
```

```
def heuristic(s):
    var1 = True
    if var1 == True:
        result = var1
    else:
        result = False
    return result
```

```
def heuristic(n):
    var1 = 1
    if var1 == 1:
        result = var1
    else:
        result = 0
    return result
```

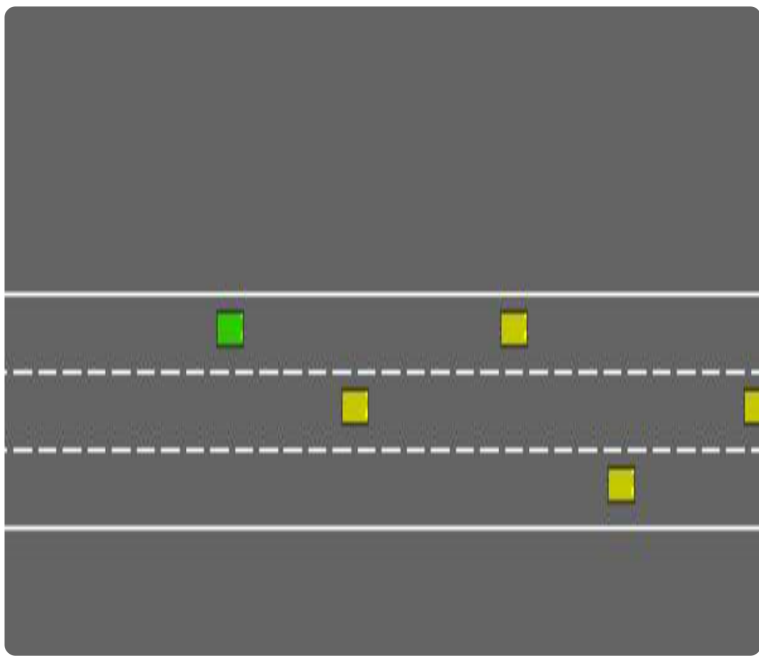
```
def func(n):
    var1 = 0
    result = 1
    if var1 == 0:
        result = 0
    return result
```

Please click "Submit" when you're ready to move on.

Submit

Figure A.2: Describing our DSL.

Step 1/2: Playing Game



The screenshot shows a top-down view of a highway with two lanes. A green car is in the left lane, and several yellow cars are in the right lane. The road has white lane markings and a dashed center line. To the right of the road are four control buttons: a yellow 'Pause' button, a red 'Reset' button, a blue 'Start Coding' button, and a purple 'Help' button.

```
Observation:  
s[0] (x): 202.83713  
s[1] (y): -0.22211969  
s[2] (x speed): 24.999996  
s[3] (y speed): 0.012383506  
s[4] (1st car x): 224.94484  
s[5] (1st car y): 4.0  
s[6] (1st car x speed): 22.541086  
s[7] (1st car y speed): 0.0  
s[8] (2nd car x): 252.98553  
s[9] (2nd car y): 0.0  
s[10] (2nd car x speed): 23.384562  
s[11] (2nd car y speed): 0.0  
Reward: 1.2666666666666666  
Time Remaining (min): 10
```

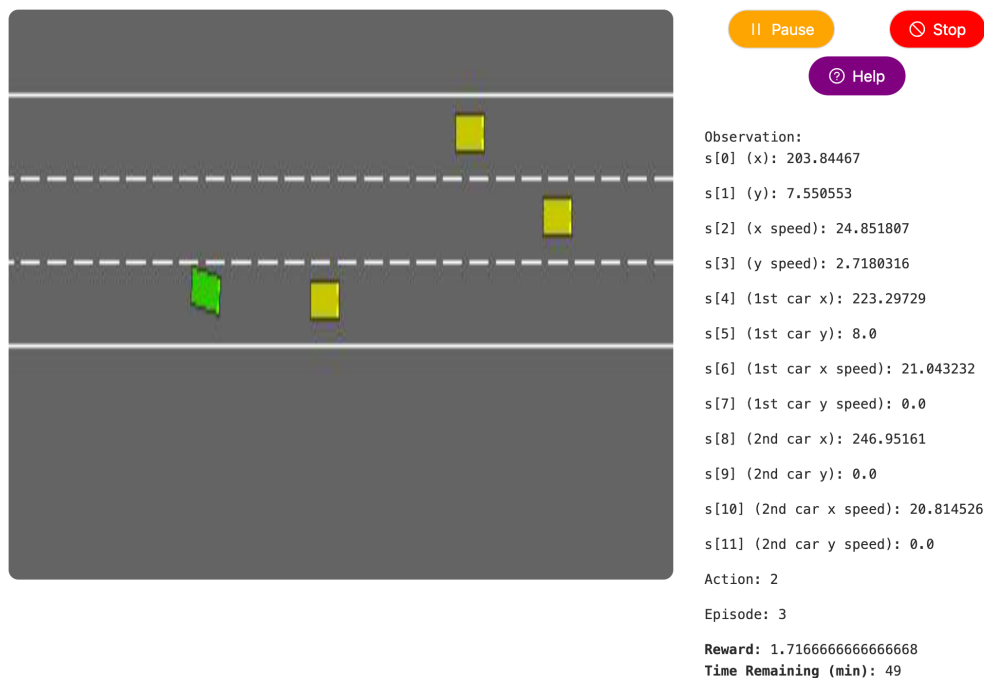
Figure A.3: Playing Highway for at most 10 minutes before start coding. At each step of the game, the system shows observations at that step and time remaining. The Help button refers to a page that contains game and language description details.

Step 2/2: Coding Your Strategy

```

3
4     env: The environment
5     s (list): The state. Attributes:
6         s[0] is your x position, approximate range is [100, 400]
7         s[1] is your y position, approximate range is [0, 8]
8         s[2] is your x speed, approximate range is [10, 30]
9         s[3] is your y speed, approximate range is [-3, 3]
10        s[4] is 1st car x position, approximate range is [100, 400]
11        s[5] is 1st car y position, approximate range is [0, 8]
12        s[6] is 1st car x speed, approximate range is [10, 30]
13        s[7] is 1st car y speed, approximate range is [-3, 3]
14        s[8] is 2nd car x position, approximate range is [100, 400]
15        s[9] is 2nd car y position, approximate range is [0, 8]
16        s[10] is 2nd car x speed, approximate range is [10, 30]
17        s[11] is 2nd car y speed, approximate range is [-3, 3]
18
19     returns:
20         action: The heuristic to be fed into the step function defined below to
21                 determine the next step and reward.
22
23     """
24     # left lane = 0, Nop = 1, right lane = 2, faster = 3, slower = 4
25     action = 0
26     """ please write your code here
27     # example code for highway game
28     # if my car (green) is not in the right-most lane, go to the right-most lane
29     if s[1] < 8:
30         action = 2
31     else:
32         action = 1
33     return action

```



Observation:
 s[0] (x): 203.84467
 s[1] (y): 7.550553
 s[2] (x speed): 24.851807
 s[3] (y speed): 2.7180316
 s[4] (1st car x): 223.29729
 s[5] (1st car y): 8.0
 s[6] (1st car x speed): 21.043232
 s[7] (1st car y speed): 0.0
 s[8] (2nd car x): 246.95161
 s[9] (2nd car y): 0.0
 s[10] (2nd car x speed): 20.814526
 s[11] (2nd car y speed): 0.0

Action: 2
 Episode: 3
 Reward: 1.7166666666666668
 Time Remaining (min): 49

Figure A.4: Writing a program to play Highway with maximum reward. Participants can evaluate their programs on the game and see the observation at each step and the action their programmatic policies take, along with a timer that shows how much time remains. The Help button refers to a page that contains game and language description details. Participants can pause whenever they want and analyze the current observation and action. Also, they can stop the evaluation process at any time, providing an opportunity for a quick debug.

Questionnaire

Please select one answer for each of the following questions

When you finish the questionnaire, please click "Submit" at the bottom

1. How old are you? (2 digit numbers. E.g. 18)

2. How do you identify yourself?

- Female
- Non-binary
- Male
- Other
- I do not wish to answer

3. Do you have a degree in computer science or related field (e.g., Information Systems and Electrical Engineering)?

- Yes
- No
- I do not wish to answer

4. What is the highest level of education you have completed?

- some secondary education (high school)
- completed secondary education (graduated high school)
- some undergraduate education (college or university)
- completed undergraduate education
- some postgraduate education
- completed postgraduate education (masters or doctorate)
- I do not wish to answer

5. How many years of programming experience do you have?

- 0-1
- 1-2
- +2
- I do not wish to answer

6. How many courses in Artificial Intelligence have you taken?

- 0
- 1
- 2 or more
- I do not wish to answer

7. Have you done any form of game AI research?

- Yes
- No
- I do not wish to answer

8. How often do you play computer games?

- Never
- Rarely
- Occasionally
- Often
- I do not wish to answer

9. How hard did you find writing a program with the language we introduced? (5 means very hard)

- 1
- 2
- 3
- 4
- 5
- I do not wish to answer

10. How hard did you find writing a program to solve the target problem? (5 means very hard)

- 1
- 2
- 3
- 4
- 5
- I do not wish to answer

11. How much do you know about python programming? (5 means very expert)

- 1
- 2
- 3
- 4
- 5
- I do not wish to answer

Submit

Figure A.5: Survey and demographics questions.

Appendix B: Original Representative Program

We show a representative program written by a participant of our study for Highway and refer to it as p , in Chapter 5. Figure 5.11 shows the original version of POLIS's improved program for p .

```
1 def heuristic(o):
2     action = 0
3     if o[1] and o[3]:
4         action = 4
5     elif (o[5] == o[1] and o[5] - o[1] <= 200) or (o[9] == o[1]
6         and o[9] - o[1] <= 200):
7         if o[1] == o[5]:
8             if o[1] < 7.9317:
9                 action = 2
10            else:
11                action = 0
12        else:
13            action = 2
14    else:
15        action = 1
16    return action
```

Figure B.1: Original POLIS's improved program for the program written by a participant of our study, which is shown in the main paper.

Appendix C: DQN Hyperparameters

We trained an agent with a two layers neural network, each includes 64 neurons, using vanilla DQN algorithm for 2000 episodes with a decaying epsilon policy and batch size = 64. Other hyperparameters are: starting $\epsilon = 1.0$, and ϵ -decay rate = 0.99, and ϵ -min = 0.01, and $\gamma = 0.99$. The learning rate is 0.0005 and the optimizer is ADAM. We trained 50 agents with different random initialization to do the restart approach explained in section 5.2.4.