Leveraging Large Language Models for Speeding Up Local Search Algorithms for Computing Programmatic Best Responses

by

Quazi Asif Sadmine

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

Master of Science

Department of Computing Science

University of Alberta

Abstract

Despite having advantages such as generalizability and interpretability over neural representations, programmatic representations of hypotheses and strategies face significant challenges. This is because algorithms writing programs encoding hypotheses for solving supervised learning problems and strategies for solving games must conduct searches in very large and discontinuous search spaces—the spaces programming languages induce. Previous studies have introduced self-play algorithms to learn programs that encode game-playing strategies. These algorithms compute a sequence of approximate best responses against target strategies. In this dissertation, we introduce a new approach that leverages the ability of large language models (LLMs) to write computer programs to provide initial candidate solutions in the programmatic space for best responses. These candidates can be a best response or serve as a seed to begin the search for a best response. Empirical results in three games that are challenging for programmatic representations show that LLMs can speed up local search and facilitate the synthesis of strategies.

Preface

I am excited to present this dissertation titled "Leveraging Large Language Models for Speeding Up Local Search Algorithms for Computing Programmatic Best Responses", which showcases my original and unpublished research conducted under the supervision of Dr. Levi Lelis.

Given the collaborative nature of this project, the inclusive pronoun 'we' is employed throughout this document. However, I accept complete responsibility for any technical or presentational errors that may occur.

Quazi Asif Sadmine

May, 2024

To my parents, sister, and nephew
Who sacrificed living miles away so I could chase my dreams

So, surely with hardship comes ease.

– Al Quran (94:5-6)

Acknowledgements

I express my deepest respect and gratitude to my supervisor, Dr. Levi Lelis. Before starting my official work under his guidance, I had limited knowledge of this research area. Through his course, he sparked in me a profound interest and dedication to this field of research. Throughout my Master's program, he has served as an exceptional mentor, always available to assist me in overcoming my weaknesses and filling my knowledge gaps. His patience and tireless work ethic have greatly influenced my research endeavors. Working with him has been an invaluable learning experience. Additionally, I would like to express my gratitude to Dr. Hendrik Baier for his valuable advice throughout this work.

I also express my gratitude to my research colleagues, Rubens O. Moraes, Habib Rahman, Thirupathi Reddy, Tales Henrique Carvalho, Kenneth Tjhia, David S. Aleixo, Zahra Bashir, Zaheen Ahmad, Mahdi Alikhasi, Mahdieh Mallahnezhad, Parnian Behdin, Amirhossein Rajabpour, Jake Tuero, and Paul Saunders for constantly supporting and guiding me throughout this journey. My friends here in Canada, including Samiul Anwar, Salima Sharmin Khan, Rayhan Kabir, Alif Rahman, Nashid Rahman, Sk Abdul Alim, and Abrar Fahim, have also played an important role in this journey.

The University of Alberta and the Alberta Machine Intelligence Institute (Amii) have played a crucial role in supporting me with funds and resources. This research could not have even begun without the support of these institutions, for which I am deeply grateful.

Finally, I extend my heartfelt thanks to my parents and sister for their unwavering love and support. Your immense love and support have been the driving force behind my success.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Programmatic representations have received significant attention due to their ability to generalize to unseen scenarios (Inala et al., 2020) and their interpretability, which allows users to understand and possibly modify the solutions (Aleixo & Lelis, 2023; Verma et al., 2018a). Programmatic representations have been used to address various problems, including program synthesis (Ameen & Lelis, 2023; Barke et al., 2020; Gulwani, 2011, 2016; Odena et al., 2021; Shi et al., 2022) and sequential decision-making problems (Aguas et al., 2018; Bastani et al., 2018; Bonet et al., 2010; Liang et al., 2023; Moraes et al., 2023).

The difficulty in using programmatic representations lies in searching through large and discontinuous program spaces, typically defined by a domain-specific language (DSL). A common approach to seeking programmatic representations of hypotheses to address supervised learning problems involves learning a function that guides the search within the program space. (Alur et al., 2013; Balog et al., 2017; Odena & Sutton, 2020; Solar-Lezama, 2009). This function is learned in a self-supervised manner, leveraging the structure of the language. For instance, in program synthesis, given input-output pairs, one must find a program that maps each input to its output (Gulwani, 2011). A guiding function can be learned by generating random programs from the language and creating training problems with random inputs, resulting in target outputs. The guiding function is then trained to solve these problems.

However, a challenge arises when this self-supervised method for learning

guiding functions is employed to solve sequential decision-making problems, particularly in the multi-agent scenario, where the agent improves its skill in the game through a self-play process. This is because the distribution of "problems" the algorithm needs to learn to solve is only discovered during self-play, making it difficult to learn a guiding function before self-play.

To better understand why the agent only knows the distribution of problems it needs to solve after learning has started, consider the following example of a self-play process for a two-player zero-sum game titled "Poachers and Rangers" (Moraes et al., 2023). In this game, poachers can attack the gates of a national park, and rangers need to defend them. The DSL defining the programmatic search space includes one or more "`attack(gate number)`" instructions for poachers and one or more "`defend(gate number)`" instructions for rangers. The self-play algorithm known as Iterated-Best Response (IBR) considers a current strategy for each player, which is used as the "target", i.e., the strategy to be defeated. The target strategy for rangers is the most recent strategy played by the poachers, and vice-versa. Suppose, initially, that the poachers' strategy is randomly determined as "`attack(3)`". The rangers must then find the best response to it (for now, we can consider a best response a strategy that is able to "defeat" the target strategy). The best response is approximated by employing a local search algorithm (e.g. stochastic hill climbing) that randomly initializes at a point in the programmatic space and looks for the strategy in its neighbors that can defeat the target strategy. In this example scenario, a best response could be "`defend(1) defend(3)`"—any strategy that defends gate `3` is a best response, as the rangers will successfully defend the national park from the poachers attack (more information about this game is given in Chapter 5). After computing such a best response to the target strategy, this new strategy of the rangers becomes the target strategy for the poachers, and the iterative process continues (Lanctot et al., 2017). The sequence of best responses that needs to be computed is only known after learning starts, making it difficult to learn a guiding function a priori.

Due to our inability to learn a guiding function a priori, the challenge of computing best responses arises in complex game environments with very large

search spaces. Often, the self-play process takes a long time to converge into a dominant strategy due to the absence of any guidance to the local search algorithms approximating best responses. Considering the ability of LLMs to write computer programs, we propose the incorporation of an LLM within the local search framework, which can perform the task of a guiding function. We chose to use the LLM only in the initialization of the search, as opposed to in every step of the search, due to its computational cost.

Hypothesis We hypothesize that the inclusion of an LLM may enable the search to initiate at a more promising part of the search space, leading to faster computation of the best response compared to existing methods.

Thesis Statement In this dissertation, our aim is to decide whether the general knowledge embedded within a language model can assist in guiding the local search within discontinuous programmatic search spaces, speeding up the computation of approximated best responses to a target strategy in the context of zero-sum two-player games.

1.1 Contributions

The dissertation introduces Local Search with LLM (LS-LLM), an integration of an LLM into local search with the goal of speeding up the search for approximate best responses within the programmatic space. The "general knowledge" of an LLM is encoded into programmatic strategies the model generates that are either a best response to a target strategy or close in the search space to a best response, which could ease the process of searching for a programmatic best response. As input, the LLM takes a description of the game, a context-free grammar describing the domain-specific language, an explanation of the domain-specific functions used in the grammar, and a programmatic strategy for which a best response must be approximated. The LLM writes a program that encodes a possible best response to the target strategy in the language described by the context-free grammar. If the strategy the LLM writes is not

the best response, it is used as a seed to initiate the search in the programmatic space for an approximated best response.

LS-LLM has been evaluated in three deterministic two-player zero-sum games that are challenging for current systems that generate programmatic strategies: Poachers & Rangers (PR), Climbing Monkey (CM), and MicroRTS. The results provide strong evidence that LS-LLM outperformed the baselines considered in the experiments. The dissertation also shows that LS-LLM's performance drops when the name of the functions and terminal symbols used in the domain-specific language are encrypted (i.e., we replace the meaningful names of the functions with meaningless names), suggesting that the knowledge the LLM leverages comes from interpreting those names. Additionally, our results show evidence that a feedback system we developed for the LLM while it attempts to generate a best response to the target strategy is an important factor in achieving better results.

This dissertation is organized as follows: Chapter 2 offers a comprehensive background on key concepts, including problem definition, domain-specific languages, abstract syntax trees, self-play learning algorithms, and local search algorithms. Chapter 3 comprises related works that describe the relationship between our work and existing literature. These concepts are essential for understanding the subsequent discussions. Chapter 4 introduces our proposed approach—Local Search with LLM (LS-LLM), providing the algorithm, a detailed step-by-step explanation, and an example for a comprehensive understanding of its functionality. Chapter 5 presents the empirical methodologies, which include three distinct problem domains, baseline systems, language models and prompts, and the details of the experiments performed, along with all relevant specifications. Chapter 6 illustrates and explains the results of our experiments. Chapter 7 concludes the dissertation with a summary of the contributions. It also outlines future directions for research that can be explored in subsequent studies. The appendix of this dissertation includes supplementary materials containing all information required to reproduce the results presented in this dissertation.

Chapter 2

Background

2.1 Problem Definition

Let $G = (P, S, s_{\text{init}}, A, T, U)$ be a deterministic two-player zero-sum game. Here, $P = \{i, -i\}$ is the pair of players, S is the set of states, and $s_{\text{init}} \in S$ is the initial state. A is the set of actions where $A_i(s) \subseteq A$ and $A_{-i}(s) \subseteq A$ are the actions taken by players i and $-i$ respectively, either alternatively or simultaneously (e.g. real-time strategy games) from any state $s \subseteq S$. This action is chosen by a deterministic strategy, which is a function σ_j mapping states to actions. A function σ_j maps states to actions, hence determining this action through a deterministic strategy. A state and an action for each player is sent to a deterministic successor function T, which returns the next state of the game. The function U determines the utility value of the terminal states. $U_i(s) = U_{-i}(s)$, since G is a zero-sum game. The objective for player i is to maximize the utility value, while the objective for player $-i$ is to minimize it. We also utilize the function U to represent the value of a given state from the point of view of player i, based on the strategies employed by the players. For instance, if player i and $-i$ follow strategies σ_i and σ_{-i} respectively starting at state s, then $U(s, \sigma_i, \sigma_{-i})$ is the value of the terminal state. We consider programmatic strategies that encode σ_i and σ_{-i} in programs written in a domain-specific language.

2.1.1 Domain-Specific Language & Abstract Syntax Tree

A domain-specific language (DSL) is defined to write programmatic strategies. The set of programs a DSL accepts is defined as a context-free grammar (M, Ω, R, I), where M, Ω, R, and S are the sets of non-terminals, terminals, the production rules of the grammar, and the grammar's initial symbol, respectively. Figure 2.1 shows an example of a DSL, where $M = \{I, C, B\}$, $\Omega = \{c_1, c_2, b_1, b_2, \text{if}, \text{then}\}$, R are the production rules (e.g., $C \to CC$), and I is the initial symbol.

$I \to \text{if}(B) \text{ then } C$
$C \to C\,C \mid c_1 \mid c_2$
$B \to b_1 \mid b_2$

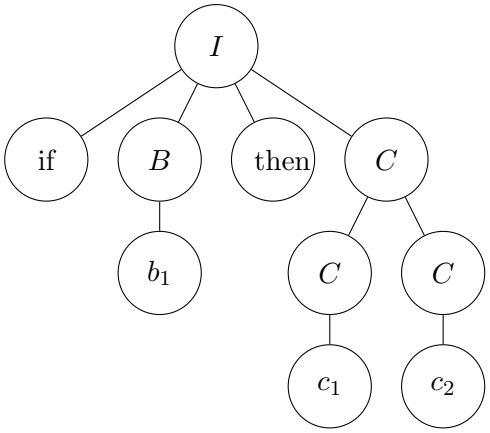


Figure 2.1: An example DSL and the abstract syntax tree for program "if b_1 then c_1 c_2" written in that DSL

A DSL D defines the possibly infinite space of programs $[\![D]\!]$, which we call programmatic search space, or programmatic space. Each program p in $[\![D]\!]$ represents a programmatic strategy of the game G.

The programs are represented as abstract syntax trees (AST). The initial symbol is the root of the tree, the non-leaf nodes are non-terminals, and the leaf nodes are the terminals of the grammar. Figure 2.1 shows an example of an AST for program "if b_1 then c_1 c_2". The terminal symbols forming the program are given by a left-to-right traversal of the leaf nodes of the tree.

Given all the strategies defined in the programmatic space, we would like to find those that satisfy the following equation.

$$\max_{\sigma_i \in [\![D]\!]} \min_{\sigma_{-i} \in [\![D]\!]} U(s_{\text{init}}, \sigma_i, \sigma_{-i})\,. \tag{2.1}$$

Strategies σ_i and σ_{-i} that satisfy Equation 2.1 represent a Nash equilibrium

profile in the programmatic space. Learning algorithms approximate a solution to Equation 2.1. The profile (σ_i, σ_{-i}) is a Nash equilibrium profile if σ_i is a best response to σ_{-i} and σ_{-i} is a best response to σ_i.

2.2 Learning Algorithms

Learning algorithms are required to conduct the self-play process that approximates the solution to Equation 2.1. There are a few learning algorithms such as Iterated Best Response (IBR) (Lanctot et al., 2017), Fictitious Play (FP) (Brown, 1951), Double Oracle (DO) (McMahan et al., 2003), and Local Learner (2L) (Moraes et al., 2023). In this section, we will explain the different approaches of IBR, FP, and 2L to learn programmatic strategies in an example domain. These three learning algorithms have been selected for explanation due to their utilization in our empirical evaluations.

We choose the game called Poachers & Rangers (PR) as an example domain to explain the learning algorithms. PR is a two-player zero-sum game without ties, where two players, called poachers and rangers, compete to win. Rangers are trying to defend the gates of a national park, whereas, poachers are attempting to enter through an undefended gate. Rangers receive the utility of 1 if they protect all attacked gates and -1 otherwise. In this chapter, we consider games with six gates.[1]

2.2.1 Iterated Best Response (IBR)

IBR is the simplest of all the learning algorithms. IBR starts with an arbitrary strategy $\sigma_0 \in [\![D]\!]$ for one of the players, for example $-i$, and approximates a best response to σ_0 for the other player i: $\sigma_1 = \text{argmax}_{\sigma_i \in [\![D]\!]} U(s_{\text{init},\sigma_i,\sigma_0})$. Then it approximates a best response to σ_1 for player $-i$: $\sigma_2 = \text{argmin}_{\sigma_{-i} \in [\![D]\!]}$ $U(s_{\text{init},\sigma_1,\sigma_{-i}})$. This process is repeated a number of times n, which is normally determined by a computational budget. The last resulting strategies σ_n and σ_{n-1} are returned as IBR's approximate solution to Equation 2.1.

In PR, if the previous strategies of poachers are `attack[1]`, `attack[4]`,

[1] A more comprehensive description about PR can be found in Chapter 5

`attack[6]`, where the action to attack the n-th gate is "attack(n)", the best response for rangers will be any strategy that defends at least gate number 6. This is because IBR only considers the latest strategy while computing the best response. So, `defend[6]` can be one of the possible best responses for rangers in this scenario. Similarly, in the next iteration, the best response for poachers will be a strategy that attacks any gates other than the gate number 6.

Achieving a dominant strategy with the help of IBR is rarely possible. In the above example, the dominant strategy for rangers is to defend all the gates starting from 1 upto n. In the above example, approximating `defend[6]` as the best response to the target strategy `attack[6]` is way easier than finding the dominant strategy (which is also a best response to the target strategy) in the search space. This is due to the fact that the AST of `defend[6]` is much smaller than the AST of the dominant strategy. So it is easier to find for the local search algorithm. So in this scenario, the poachers may respond to `defend[6]` with `attack[5]`, leading the rangers to counter with `defend[5]`, followed by the poachers responding with `attack[6]`, resulting in a loop of best responses. Hence, in practice, IBR tends to converge very slowly.

2.2.2 Fictitious Play (FP)

Unlike IBR, FP takes into account all the previous strategies of the opponent when approximating a best response. This makes FP capable of providing better search signals compared to IBR. Additionally, there is no possibility of creating a loop of best responses.

In FP, σ_2 is the same as IBR, i.e., $\sigma_2 = \text{argmin}_{\sigma_{-i} \in [\![D]\!]} U(s_{\text{init},\sigma_1,\sigma_{-i}})$. But from σ_3 of player i, it considers a target strategy σ_t, which is a combination of σ_0 and σ_2 of player $-i$. Similarly, the computation of σ_4 of player $-i$ requires a target strategy σ_t, which is a combination of σ_1 and σ_3 of player i. This implies that in FP, the best response for player i will always require a target strategy that combines all the previous strategies played by player $-i$, and vice versa.

Table 2.1 shows an example of the learned strategies of Poachers & Rangers

Poachers	Rangers
attack[1]	defend[1]
attack[3]	defend[1,3]
attack[6]	defend[1,3,6]

Table 2.1: Illustration of Fictitious Play (FP) for the Poachers & Rangers (PR) Domain

after three iterations. In the first iteration, poachers attacked gate number 1, prompting rangers to defend that gate as the best response to the poachers' strategy. In the next iteration, poachers attacked gate number 3 as the best response to `defend[1]`. Consequently, rangers defended gates number 1 and 3 as the best response to all previous strategies of the poachers, i.e., `attack[1]` and `attack[3]`. In the third iteration, poachers attacked gate number 6 as the best response to `defend[1]` and `defend[1,3]`. As the best response, rangers defended all three gates attacked by poachers, i.e., gates number 1, 3, and 6.

In FP, the best response is computed for a mixture of all strategies computed for the other player. For example, in the iteration in which FP returns `defend[1,3,6]`, this best response is for the strategy that plays `attack[1]`, `attack[3]`, or `attack[6]` with equal probability (i.e., $\frac{1}{3}$ probability in this case). This type of strategy is known as a mixed strategy σ. The strategies that can be played with a non-zero probability are said to be in the support of σ.

2.2.3 Local Learner (2L)

In 2L, in contrast to FP that uses all previous strategies encountered in learning, only the strategies that are "useful" to compute a best response are considered. The determinant factor here is whether a strategy is useful or not. 2L considers a strategy σ useful if σ provides information that is not provided by any other strategy in the set, as we will explain in the following example.

Table 2.2 illustrates the learned strategies of poachers and rangers after three iterations of 2L. If we compare Table 2.1 and Table 2.2, we find that the difference is the absence of `defend[1]` and `defend[1,3]`. This means, these two strategies are not useful in the context of 2L. The rea-

son is, `defend[1,3,6]` is already a superset of the strategies `defend[1]` and `defend[1,3]`. This implies that the best response to `defend[1]`, `defend[1,3]`, and `defend[1,3,6]` will have the same constraints as the best response to only `defend[1,3,6]`.

Poachers	Rangers
`attack[1]`	`defend[1,3,6]`
`attack[3]`	
`attack[6]`	

Table 2.2: Illustration of Local Learner (2L) for the Poachers & Rangers (PR) Domain (Moraes et al., 2023)

Approximating best responses can be a costly operation in complex domains when highly informative learning algorithms are used, such as FP. This is because one needs to play a potentially large number of games for every candidate best response evaluated. In the case of FP, each candidate is evaluated against all previous strategies encountered in learning. Even in a simple domain such as PR, 2L is able to reduce the overall computational cost because it is selective with respect to the set of strategies against which one needs to evaluate candidate solutions. In more complex domains, 2L can have a large impact in terms of sample efficiency of the learning process. 2L tends to perform better than IBR, FP, and DO (Moraes et al., 2023), and this explains our choice for it in our experiments.

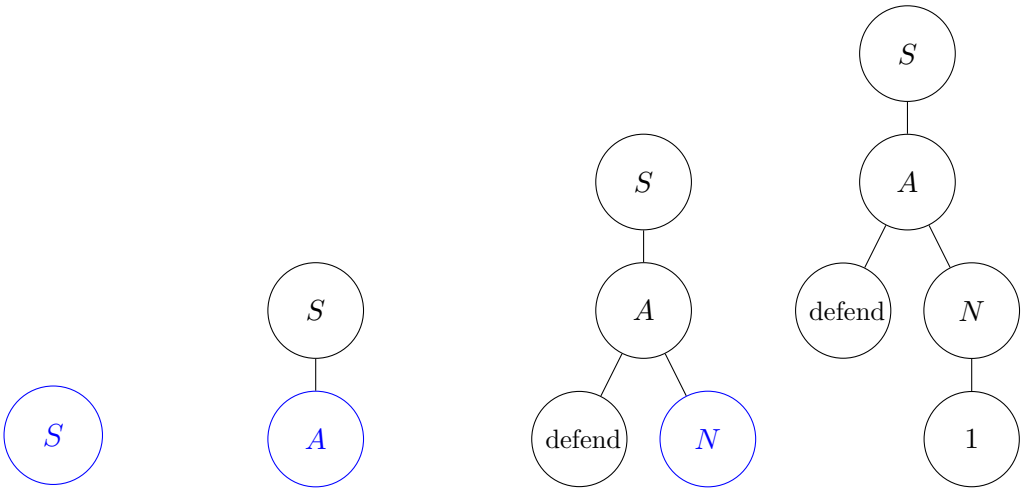
2.3 Searching for Programmatic Best Responses

The computation of a programmatic best response to a strategy is conducted within the programmatic search space defined by the DSL. We consider local search algorithms for approximating programmatic best responses. In particular, we use an implementation of stochastic hill-climbing (SHC). Given a target programmatic strategy σ, SHC starts with an arbitrary strategy σ' as a candidate for a best response to σ. In SHC, an initial candidate σ' from which the search is initialized using the following process. We start with the initial symbol S of the DSL and apply a production rule to replace S; the

rule is chosen uniformly at random. One of the non-terminal symbols in the resulting string is then replaced with the symbols on the right-hand side of a production rule that is also chosen uniformly at random. This process continues until a string with only terminal symbols is generated. Figure 2.2 depicts an example of the process for generating an initial candidate for the search. The image shows how the AST is built after the production rules $S \to A$, $A \to \text{defend}(N)$, and $N \to 1$ are applied.

$$
\begin{aligned}
S &\to A \\
A &\to A\,A \mid \text{defend}(N) \\
N &\to 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6
\end{aligned}
$$

(a) An example DSL for rangers



(b) Synthesis starting from initial symbol S

(c) S is expanded to non-terminal symbol A

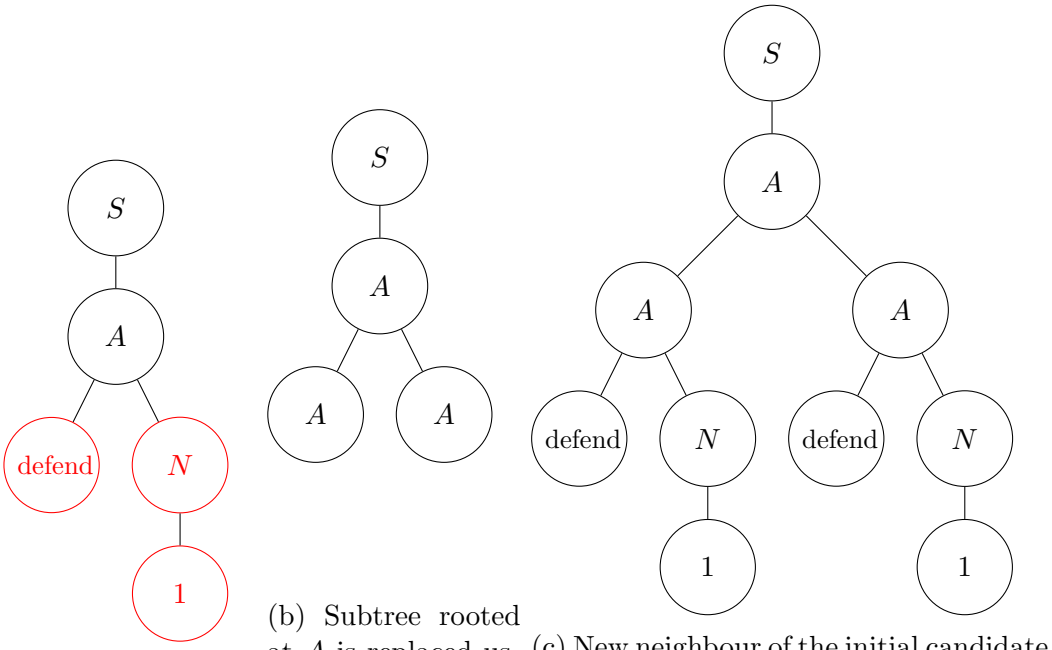
(d) A is expanded to production rule defend(N)

(e) N is expanded to terminal symbol 1

Figure 2.2: An example of synthesizing a programmatic strategy using the production rules of the DSL

SHC searches a space defined by a neighborhood function $\mathcal{N}_k(\sigma)$ that receives a candidate σ and returns a set of k strategies—the neighbors of σ in the search space. SHC evaluates all k neighbors in terms of U-value and selects the best neighbor of σ. For example, if we are computing a best response to a strategy of player $-i$, then we need to find a σ that maximizes U, so SHC

selects the neighbor whose U is the largest. This process is repeated with the newly selected candidate solution. SHC stops its search if none of the neighbors has a U-value that is better than the current candidate, i.e., the algorithm reaches a local optimum. We implement SHC with a restarting strategy: once SHC reaches a local optimum, we restart the search from another randomly chosen initial candidate solution. SHC with restarts returns the best solution encountered across multiple runs of the search.



(a) Initial candidate

(b) Subtree rooted at A is replaced using another production rule

(c) New neighbour of the initial candidate

Figure 2.3: An example of generating a neighbour from a given initial candidate

SHC is stochastic because the selection of the initial candidate and the neighborhood function are stochastic. The neighbors $\mathcal{N}_k$ returns are generated as follows. Each of the k neighbors is generated by randomly choosing a node n in the AST of the program σ that represents a non-terminal symbol (e.g., the nodes representing S or A in Figure 2.2) and replace the subtree rooted at n with a randomly generated sub-tree. This new sub-tree is generated by following the same process described to generate the initial candidate. For

instance, in Figure 2.3, the subtree rooted at A is replaced by non-terminal symbols $A\,A$ following the production rule of A. Then the A on both sides are expanded until terminal symbols are reached.

Although other local search algorithms can be used, such as Simulated Annealing (Husien & Schewe, 2016; Kirkpatrick et al., 1983), we use SHC in our experiments because previous work showed that it performs well with programmatic strategies (Carvalho et al., 2024; Moraes et al., 2023). Moreover, the idea of generating the initial candidate of the search with an LLM can, in principle, be used with any local search algorithm.

Chapter 3

Related Work

Our work intersects with several areas, including program synthesis, programmatic reinforcement learning, the application of LLM in game domains, synthesis of programmatic policies, and optimization. This chapter aims to explore the connections between our work and these areas, highlighting the unique aspects of our approach compared to existing literature.

3.1 Synthesizing Programmatic Strategies in Games

There is a growing interest in the synthesis of programmatic strategies in games. Previous works have demonstrated success in generating programs to solve reinforcement learning (RL) problems (Inala et al., 2020; Qiu & Zhu, 2022; Trivedi et al., 2021; Verma et al., 2018b). Our work is closely related to these RL studies. The synthesis of programs for game-playing strategies also extends to puzzle and logic games, with an SMT solver employed for single-player puzzles (Butler et al., 2017) and a SAT solver for logic games (Farzan & Kincaid, 2018). In multi-agent settings, some studies have focused on extracting interpretable policies from trained neural models (X. Liu et al., 2023; Milani et al., 2022). Closest to our work, various approaches have been used to find strategies for two-player zero-sum games—from genetic programming (Mariño & Toledo, 2022), to Monte Carlo Tree Search (Medeiros et al., 2022) and local search (Mariño et al., 2021; Moraes et al., 2023).

3.2 Application of LLMs in Games

Previous work has leveraged LLMs in games, but not for generating programmatic strategies. They have been used to perceive, plan, and act (Park et al., 2023), often decomposing long-term goals into subtasks and monitoring their execution (Z. Wang et al., 2023), and/or integrating additional agent features such as memory (Zhu et al., 2023) and/or automatic learning curricula (G. Wang et al., 2023). Much of this work has focused on online planning in open-world games such as Minecraft or Sims-like social simulations. Our work aims at the offline generation of programmatic strategies to play games that are not modified or re-planned in an online fashion during execution.

3.3 Synthesizing Programmatic Policies in Non-Game Domains

Some previous work has used LLMs for the synthesis of programmatic policies, but not for games. In generalized planning (GP), LLMs have shown much promises (Silver et al., 2023) where the objective is to synthesize programs that solve classical planning problems (Celorrio et al., 2019). One of the very recent works has employed LLMs in the code generation for robot decision-making (Liang et al., 2023; Singh et al., 2023). In contrast to our work, these fields have so far considered single-agent problems, while we use LLMs for learning zero-sum two-player games. However, it should be noted that the generation of best responses, which we use iteratively in self-play, can be viewed as a single-agent problem, as it involves keeping the strategies of all agents except one constant in each iteration.

3.4 Application of LLMs in Search and Optimization

Outside the context of competitive games, another way to view our work is as using LLMs to guide local search within a programmatic search space. Some studies have explored integrating LLMs into different search approaches.

LLMs have been used for optimization in a simple interactive prompting loop that asks for new solutions based on past solutions and their values (Guo et al., 2023; Yang et al., 2023); they have been used to guide Monte Carlo Tree Search by providing a common-sense world model and a heuristic policy (Zhao et al., 2023); and LLMs have been integrated as selection, crossover, and/or mutation operators into a variety of evolutionary algorithms (Chen et al., 2023; Lehman et al., 2022; S. Liu et al., 2023; Meyerson et al., 2023), including multi-objective (F. Liu et al., 2023) and quality-diversity EAs (Nasir et al., 2023). To our knowledge, we are the first to integrate LLMs into local search. Also note that most prior work is relatively time-consuming and resource-intensive (F. Liu et al., 2023) due to calling the LLM in every step of the search, while we aim at reducing LLM calls by seeding/initializing a traditional (local) search algorithm with it. Trade-offs between computational cost and performance could be explored in future work by varying the frequency of LLM calls.

3.5 Other Related Works

Some recent studies have demonstrated techniques for generating domain-specific languages (DSL) using LLMs (Jain et al., 2023; B. Wang et al., 2023). In our approach, instead of generating the DSL with LLMs, we provide a prewritten DSL and tasked the LLM with writing programs that follow the DSL's constraints. In a separate study, closed-source LLMs were analyzed in the context of data contamination (Balloccu et al., 2024). The study demonstrated the potential for iterative improvement of LLMs using data extracted from the prompt. Our work is related to this study involving closed-source LLMs, as there is a potential concern that these models may be trained on data embedded within the prompt, demanding further investigation.

Chapter 4

Local Search with LLM (LS-LLM)

In this chapter, we introduce Local Search with LLM (LS-LLM) with the goal of speeding up the local search for finding programmatic best responses. LS-LLM benefits from the ability of LLMs to encode common knowledge derived from its training data. This includes the skill of writing computer programs that can potentially speed up the search to find programmatic best responses. LS-LLM first requests a best response from the LLM to a target strategy. If the LLM fails to generate one, LS-LLM uses the LLM's output to seed the search for a best response in the programmatic space. The idea here is that, although the LLM may not directly produce a best response, its attempt to do so could result in a response that closely approximates one in the programmatic space. In the following, we describe our proposed algorithm.

Algorithm 1 shows our proposed LS-LLM. It receives a strategy σ_t for which one needs to compute a best response, the initial state of the game s_{init}, a large language model M, a budget B specifying the number of model queries that is allowed, a domain-specific language D, and a local search algorithm `LS`. LS-LLM returns an approximate best response to σ_t.

The programmatic best response is denoted as σ_{BR}. LS-LLM first requests a σ_{BR} written in D from M (line 1). If σ_{BR} is able to achieve a higher utility value than σ_t following the learning algorithm, it is returned (line 3). For instance, if $U = \{1, 0, -1\}$, when σ_{BR} is a strategy of i, we consider σ_{BR} as the best response if $U(s_{\text{init}}, \sigma_{BR}, \sigma_t) = 1$. On the other hand, for player $-i$, it

Algorithm 1 LS-LLM

Require: Target strategy σ_t, initial state s_{init}, large language model M, model budget B, domain-specific language D, local search algorithm `LS`.

Ensure: An approximate best response σ_{BR} to σ_t.

1: $\sigma_{BR} \leftarrow$ ask M for a best response to σ_t written in D
2: **if** σ_{BR} is indeed a best response to σ_t **then**
3: return σ_{BR}
4: **for** $i = 1, \cdots, B-1$ **do**
5: $\mathcal{F} \leftarrow$evaluate $U(s_{\text{init}}, \sigma_{BR}, \sigma_t)$ and return feedback
6: $\sigma_{BR} \leftarrow$ ask M for a best response to σ_t written in D while providing $\mathcal{F}$ as input to M
7: **if** σ_{BR} is indeed a best response to σ_t **then**
8: return σ_{BR}
9: **return** `LS`$(s_{\text{init}}, \sigma_t, \sigma_{BR})$

should be $U(s_{\text{init}}, \sigma_t, \sigma_{BR}) = -1$. Based on the learning algorithm, σ_t might be a mixture of several strategies. For example, FP includes in the support of σ_t all the best responses encountered in the learning process.

If σ_t represents a mixed strategy, $U(s_{\text{init}}, \sigma_{BR}, \sigma_t)$ is given by the weighted sum $\sum_{\sigma \in \sigma_t} w_\sigma \cdot U(s_{\text{init}}, \sigma_{BR}, \sigma)$. In the summation, we abuse the notation and denote σ_t as the set of strategies in its support, and w_σ is the weight of σ in the support of σ_t. The value of each $U(s_{\text{init}}, \sigma_{BR}, \sigma)$ is obtained by having σ_{BR} and σ play the game from the initial state s_{init}. In the case of a mixed strategy, the algorithm terminates early (lines 3 and 8), if $U(s_{\text{init}}, \sigma_{BR}, \sigma) = 1$ for all σ in the support of σ_t. For some strategies σ_t, the best response will not yield $U(s_{\text{init}}, \sigma_{BR}, \sigma_t) = 1$ or $U(s_{\text{init}}, \sigma_{BR}, \sigma) = 1$ for all σ in the support of σ_t. This is because sometimes the best a player can do is to draw or even lose the game. If this is the case for a given σ_t, then Algorithm 1 returns an approximated best response by searching in the space of programs while using the σ_{BR} the LLM generates as the seed to the search, as we explain below.

If M fails to generate a best response to σ_t in the first attempt, it is requested $B-1$ more times. In those attempts, LS-LLM employs the feedback information obtained from the computation of the utility $U(s_{\text{init}}, \sigma_{BR}, \sigma_t)$, which involves playing σ_{BR} against all strategies in the support of σ_t (line 5). Feedback could be provided in different forms to the LLM. We provide as

feedback some of the information from the match played between σ_{BR} and the last strategy added to the support of σ_t as feedback. We use the last strategy added to σ_t because it is the most recent strategy added to the support and is often the strongest. In our experiments, we consider two different ways of providing feedback, depending on the problem domain. These are: a sample of actions that σ_{BR} and one of its opponents (the last strategy added to the support of σ_t) took in a match between them, or a textual description of the last state of this match. If any attempt of the LLM is successful at computing a best response, then LS-LLM returns it (line 8); otherwise, the for-loop continues and the LLM gets another chance to generate a best response.

Finally, if M is not successful in generating the best response in any of the attempts, LS-LLM calls the local search function `LS` (line 9). `LS` uses the strategy σ_{BR} generated by M that has so far achieved the best utility value against σ_t as the search seed to achieve the best response to σ_t. Note that, due to the possibly large number of strategies in the support of σ_t, the utility of the strategies σ_{BR} against σ_t can vary widely.

In practice, LLMs have limitations on the size of their input. Due to this limitation, we choose to use an approximation of σ_t in the prompt. We provide only the last strategy added to the support of σ_t i.e., if σ_t is a mixed strategy of $\sigma_1, \sigma_2, \cdots, \sigma_n$, we describe σ_n as the target strategy, and request the best response from M. This is similar to assuming that all self-play algorithms can remember as much as IBR remembers: the last strategy computed to the other player. Note that this is only a limitation of the LLM's attempt to generate a best response. When using the search algorithm `LS` (line 9), the evaluation is carried out with all strategies in the support of σ_t. Even with this limitation, our empirical results show that the LLM can provide helpful initial candidates for the search for the best responses to the mixed strategies 2L considers.

In addition to helping with the computation of the best responses, we also use the LLM to generate an initial strategy (σ_0). In this case, the prompt is not conditioned on a target strategy, but we request a "strong" strategy to play the game. The details of the prompts used are in the Appendix A.

Example 1 *Consider a PR instance with 6 gates as the initial state and* $\sigma_t =$ `attack(1) attack(3)`, $B = 2$. *M is asked to generate a best response in D using an input prompt and it returns* $\sigma_{BR} =$ `defend(1) defend(2)`. *Now* σ_{BR} *is sent for verification to check if it is indeed a best response to* σ_t. *The result comes as negative, as* σ_{BR} *fails to defend gate number 3. Since* $B - 1 = 1$, $U(s_{init}, \sigma_{BR}, \sigma_t)$ *gets evaluated and returns* $\mathcal{F} =$ *"gate number 3 attacked by poachers is not yet defended". Now, M is again asked for a best response to* σ_t. *This time,* $\mathcal{F}$ *is appended to the previous input prompt. Now,* σ_{BR} *is assigned a new response* `defend(1) defend(2) defend(3)`. *Now* σ_{BR} *is sent again for verification and the result shows that it is indeed a best response to* σ_t. *Therefore,* σ_{BR} *is returned as the result.*

Chapter 5

Empirical Methodologies

In this chapter, we will describe our test domains and experimental setup.

5.1 Problem Domains

We evaluate our hypothesis in three games: Poachers & Rangers (PR) and Climbing Monkey (CM) (Moraes et al., 2023), and MicroRTS (Ontañón et al., 2018). PR and CM are challenging for current algorithms synthesizing programmatic strategies but easy for humans; MicroRTS is a challenging real-time strategy game with an annual competition.[1]

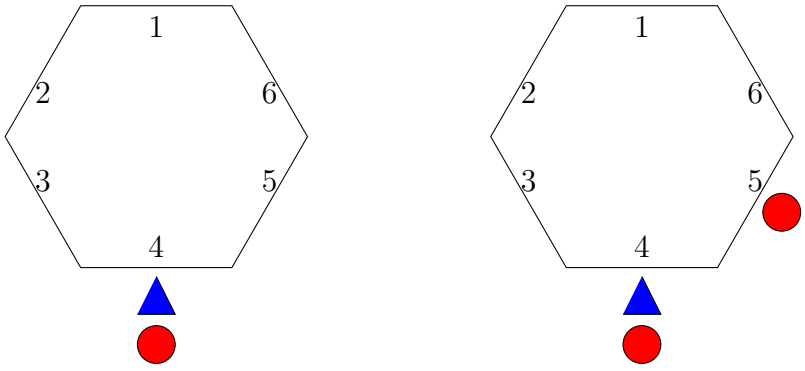
5.1.1 Poachers & Rangers (PR)

As previously mentioned in Chapter 2, PR is a two-player zero-sum game without ties, where two teams, called poachers and rangers, compete against each other. The environment consists of a national park where the rangers are trying to defend the gates of the park from inside. On the other hand, poachers are outside the park and trying to enter through the gates. Given a number of gates N, rangers win the game if they defend all attacked gates; poachers win the game if they attack an unprotected gate. One of the key features of PR is that this game is unconstrained. This is because rangers can choose to defend all N gates and poachers can choose to attack all N gates. Programs encoding strategies for rangers are of the form `defend(1)`, `defend(20)`, while encoding strategies for poachers are of the form `attack(1)`, `attack(2)`, `attack(10)`.

[1] https://sites.google.com/site/micrortsaicompetition/

Although the optimal strategy for rangers is trivially `defend(1)`, `defend(2)`, $\cdots$, `defend(N)`, the synthesis of this strategy through self-play was shown to be difficult (Moraes et al., 2023). This is because the program can be long, depending on N, and the algorithm needs to independently discover that each of the gates must be defended. The DSL for this game has been intentionally made complicated by disallowing loops, which can dramatically simplify the task. We use this domain with a DSL without loops in our experiments because it represents a challenging system for synthesizers performing search, but it can potentially be simple for systems that leverage general-knowledge systems such as LLMs.

Figure 5.1 shows a simple example of the PR environment. Here, there are six gates in total, denoted by $1, 2 \cdots, 6$. The area inside the hexagon represents the national park. The poachers are depicted as red circles, and the rangers are depicted as blue triangles. In Figure 5.1a, the rangers are winning the game as they successfully defended gate number 4 from the poachers' attack. Here, the strategy for poachers is `attack(4)`, and the strategy for rangers is `defend(4)`. In Figure 5.1b, the poachers are winning, as the rangers failed to defend gate number 5, which was attacked along with gate number 4. Here, the strategy for poachers is `attack(4) attack(5)`, and the strategy for rangers is the same as Figure 5.1a.



(a) Rangers winning to poachers (b) Rangers losing to poachers

Figure 5.1: Visual representation of the Poachers & Rangers (PR) domain

5.1.2 Climbing Monkey (CM)

CM is a game in which two monkeys compete to climb a tree with an infinite number of branches. The game concludes when a budget is reached, and the winning monkey is the one that has climbed higher than the other. If both monkeys end up on the same branch, the result is a draw. The primary constraint of this game is that a monkey must climb to branch $i + 1$ from branch i; in other words, no branch can be skipped. For example, the strategy `climb(1)`, `climb(2)`, `climb(20)` allows the monkey to reach the branch 2 of the tree; instruction `climb(20)` is ignored. Similarly to PR, the optimal strategy is trivial, but hard to achieve with current systems: `climb(1)`, `climb(2)`, $\cdots$, `climb(N)`, for a tree with N branches. The DSL of CM also prohibits loops, similar to PR. For the synthesizer, finding the optimal program in CM is a challenging task (Moraes et al., 2023), unlike for humans.

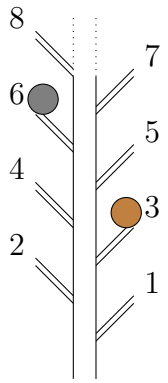


Figure 5.2: Visual representation of the Climbing Monkey (CM) domain

A simple example of the CM environment is shown in Figure 5.2. The vertical lines represent the trunk of the tree, while the dotted lines represent both infinite height and an infinite number of branches. The branches are labelled as $1, 2, \cdots, 8$. The monkeys, represented by gray and brown circles, are located in branches 3 and 6, respectively. Here, the strategy of the first monkey is `climb(1) climb(2) climb(3)`, and the strategy of the second monkey is `climb(1) climb(2) climb(3) climb(4) climb(5) climb(6)`. The second monkey is the winner, since it was able to climb more branches than the first monkey.

For both PR and CM we provide as feedback $\mathcal{F}$ to the model a textual description of the end-game state of the match played between σ_{BR} and the last strategy added to the support of σ_t. For PR, the feedback describes which gates were protected and which undefended gates the poachers attacked. For CM, the feedback reports how many branches the monkeys climbed and if they won or lost the match. The prompts with the feedback are given in the Appendix A.

5.1.3 MicroRTS

MicroRTS is a real-time strategy (RTS) game that is popularly used in artificial intelligence research for evaluating intelligent systems. The agent in this game needs to control potentially large number of units in real time, which makes the game challenging for learning and planning systems. The game is played on a gridded map as shown in Figure 5.3. This RTS game can include various units such as Bases, Barracks, Resources, Workers, Heavy, Light, and Ranged units. The size of the map can also vary, resulting in the potential requirement of different strategies. In MicroRTS, the player starts with a Base and, depending on the map, with other units. The Base allows the player to train Worker units and store resources; Workers can build structures (Base or Barracks), collect resources, and attack opponent units. Barracks can train combat units, including Light, Ranged, and Heavy units. Heavy units require more resources to train. They move at a slower pace compared to other combat units but have the highest health points, making them very challenging to eliminate. Light units, on the other hand, are the fastest but possess fewer health points than heavy units, thus placing them at a disadvantage. Although ranged units are relatively easy to eliminate due to their lower health points, their primary strength lies in their ability to attack opponents from a longer distance. This creates the possibility that they may damage or eliminate other melee units even before the latter can attack them. A player wins the game if they eliminate all units and structures of the other player.

Figure 5.3 illustrates an example scenario of the MicroRTS game. This match is played on a 9×8 gridded map, with units depicted as squares or

circles. A blue border indicates units belonging to player 1, while a red border indicates units belonging to player 2. Green cells represent neutral Resource cells, each containing 10 Resources. Squares with a light gray color represent Bases, each containing 5 Resources. Squares with a dark gray color represent Barracks. Workers are represented by circles of the same color as Barracks. Heavy, Light, and Ranged units are represented as yellow, orange, and blue circles, respectively.

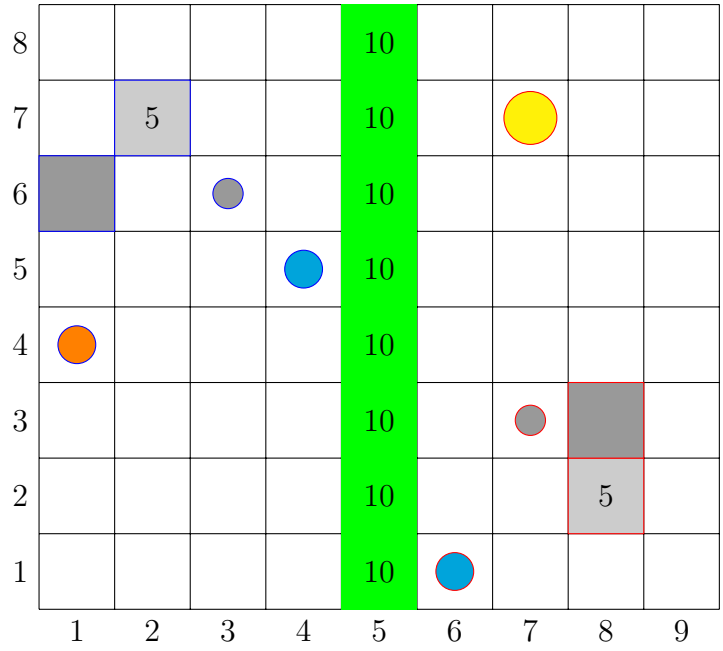


Figure 5.3: Visual representation of the MicroRTS domain

We use the following maps from the MicroRTS repository,[2] with the map size in brackets: NoWhereToRun (9×8), DoubleGame (24×24), and BWDistantResources (32×32). We use these maps because they differ in size and structure; images of the maps are provided in the Appendix A. We use GPT 3.5 in our experiments. We used a GPT 3.5 model that was trained with data collected not past January 2022, when no programmatic strategy for MicroRTS had been published online.

The programmatic strategies for MicroRTS are written in the Microlanguage, a domain-specific language developed for the game Mariño et al., 2021.

[2]https://github.com/Farama-Foundation/MicroRTS/

The Microlanguage includes functions such as `harvest` and `moveToUnit`, which allow one to encode strong strategies even in short programs. The Microlanguage uses for-loops to offer a priority scheme over actions. That is, functions called earlier in a loop have higher priority over those called later in the loop. We describe the Microlanguage in detail in the Appendix A.

MicroRTS is not a symmetric game, as it depends on the starting location of the players. For example, if strategy σ_1 defeats σ_2 when the former starts in location 1 and the latter in location 2, it does not mean that σ_1 will defeat σ_2 if we swap their initial locations. To ensure a fair evaluation, each pair of strategies plays two matches on each map, so that each player can start in each of the two initial locations on the map.

5.2 Baseline Systems

We evaluate LS-LLM with 2L as the learning algorithm and SHC as the search algorithm; i.e., we use 2L to determine the target strategies σ_t used in every iteration of the algorithm, and LS-LLM with SHC to approximate a best response to σ_t. In the results plots, we denote our system as 2L(LS-LLM). We use 2L because it was shown to outperform IBR, FP, and DO in the three games that we use in our experiments Moraes et al., 2023. Furthermore, 2L was used as a baseline in the 2023 MicroRTS competition and was second in the competition, only behind another baseline, which was a programmatic strategy written by human programmers.[3] Thus, 2L with SHC (without LLM) is our main baseline. We also use FP and IBR (also with SHC as the search algorithm) as baselines to offer a larger pool of strategies.

5.3 Language Model and Prompts

We use three different prompts for each domain: one for the initial strategy of the first iteration of 2L (called "initial"), one for the model's first attempt to generate a best response to σ_t (called "first-attempt"), and one for the

[3] https://sites.google.com/site/micrortsaicompetition/competition-results/2023-cog-results

remaining $B-1$ attempts (called "feedback-attempt"). We provide a description of the domain-specific language in all three prompts. The description is given as a context-free grammar with explanations of the functions used in the language. We also provide a description of the environment or map to the LLM. It is important to note that we do not disclose the key idea for finding the best response. For instance, in the context of CM, we do not explicitly explain that the monkey that reaches a higher branch than its opponent wins. Instead, we simply state that the goal for one monkey is to defeat the other. In first-attempt and feedback-attempt we also provide the target strategy σ_t and explain that this is the strategy that needs to be defeated. Finally, in feedback-attempt, we provide a sample of 10 actions of the model's previous attempt at a best response to σ_t issued in a match against σ_t. We do not provide all actions issued in the match due to the model's limited input length. We provide all three prompts in the Appendix.

5.4 Experiments Performed

We performed three experiments. In the first experiment, we compare 2L with LS-LLM to three other algorithms with local search: 2L, FP, and IBR on PR, CM, and three MicroRTS maps. In this experiment, we compare the different approaches in terms of gates protected (PR), branches climbed (CM), and winning rate (MicroRTS). The winning rate of a strategy is computed for a set of opponent strategies, which are given by the strategies generated by the other evaluated systems; we sum the number of victories and half the number of draws and divide this sum by the total number of matches played (Ontañón, 2017). For example, the winning rate of a strategy that wins 5, loses 2, and draws 3 matches is $\frac{5+1.5}{10} = 0.65$. The performance metric of each domain is evaluated in terms of the number of games each system needs to play to achieve a level of performance; the results are presented in plots where the y-axis shows the performance and the x-axis the number of games played (Figure 6.1 shows two examples). This experiment is meant to evaluate our hypothesis that LS-LLM can speed up, in terms of the number of games played, the computation

of best responses.

The second and third experiments are intended to improve our understanding of LS-LLM. In the second, we evaluate LS-LLM while removing from feedback-attempt the set of actions sampled from the match played between σ_{BR} and σ_t. This is to measure the impact of feedback on producing a best response or an initial candidate for search.

In the third experiment, we "encrypt" the names of the functions and terminal symbols in the Microlanguage. We still explain in the prompt what each function does, but we use meaningless names for them. For example, in the prompts used in the first and second experiments, we explain that the function `hasNumberOfUnits(T, N)` *checks if the ally player has N units of type T*. In the third experiment, this function is called `b1`, with the same explanation provided. Our goal is to verify how important the names of functions and nonterminals are to the LLM in understanding the problems and providing helpful suggestions of best responses.

5.4.1 Other Specifications

All experiments were run on computers with 2.6 GHz CPUs and 12 GB of RAM available. GPT-3.5 was used with OpenAI's API. We used a budget $B = 5$ for LS-LLM. In PR, we set the number of gates to 60 and leave the number of branches for CM unbounded. We use the value of $k = 1,000$ in the function $\mathcal{N}_k$. SHC is run with a time limit of $2,000$ seconds. Once SHC reaches a local optimum, if there is still time allowed to search, it restarts the search from the initial candidate the LLM has suggested. After reaching the time limit, the SHC with restarts returns the best program encountered across different runs. This is SHC's approximation to a best response to the strategy σ_t.

Chapter 6

Empirical Results

In this chapter, we present the results of three experiments in two distinct sections— Section 6.1 ("Programmatic Best Responses") and Section 6.2 ("Ablation Experiments"). Section 6.1 includes the results from the three domains, covering the findings of three different maps within the MicroRTS environment. Section 6.2 focuses solely on the results derived from a specific map (9×8) in MicroRTS.

6.1 Programmatic Best Responses

Figure 6.1 presents the results for PR and CM. 2L(LS-LLM) represents our contribution, which uses 2L as the learning algorithm, and LS-LLM to approximate the best responses. Note that each result in this chapter includes the average and the 95% confidence interval over 30 independent runs of the corresponding system. IBR(LS), FP(LS), and 2L(LS) are our baselines, which use IBR, FP, and 2L, respectively, with the same local search algorithm we use with LS-LLM: SHC. In this section, we refer to the algorithms as LS-LLM, IBR, FP, and 2L.

LS-LLM outperforms all baselines by a large margin. In PR, LS-LLM learns to defend all 60 gates with less than 20 games played. Interestingly, from the "initial-attempt" prompt, the agent learns to defend all the gates in most runs. In these cases, the agent was able to obtain the optimal strategy without any search involved, relying solely on the LLM. By contrast, progress for systems that rely only on search is slow. The LLM not only generates best

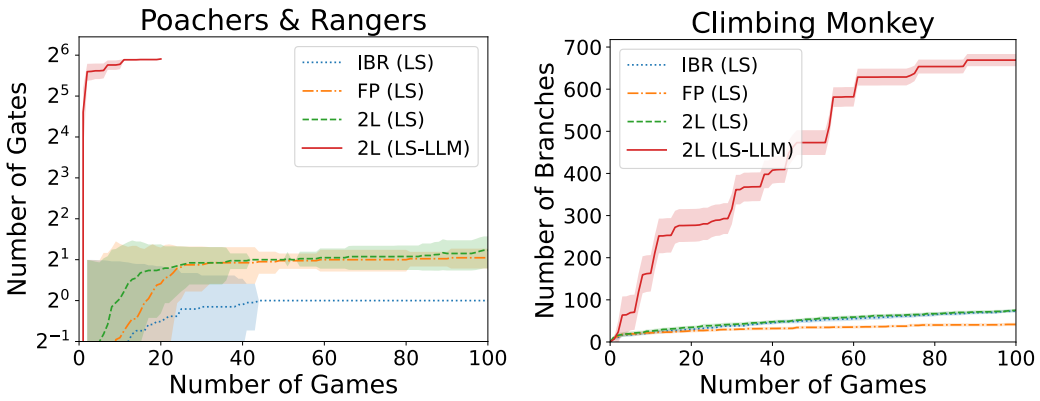


Figure 6.1: Average number of gates covered and branches climbed by the number of games played for LS-LLM, 2L, FP, and IBR in the games of PR and CM. The average and the 95% confidence interval are over 30 independent runs of the systems.

responses, but the best responses it generates cover more gates than is required to best respond to the poachers strategy. For example, if the current poachers strategy σ_p is `attack(1)`, then the simplest best response to it is `defend(1)`, which is the response an algorithm searching in the space of programs will most likely find, due to the minimal size of the program. Instead of returning this simplest best response, the LLM often returns strategies that cover more gates than needed, for example, the LLM could return "`defend(1) defend(2) defend(3)`" for σ_p, which is also a best response, but allows faster progress to the dominant strategy covering all 60 gates.

In CM, the result shows that LS-LLM was able to generate programs that climb to much higher branches than the baselines can generate. Within only 100 games, LS-LLM was able to find a strategy that allows the monkey to climb almost 700 branches, while the baselines could not even climb 100 branches. In this domain, we observe similar behavior of generating very strong best responses by the LLM. If the current strategy is `climb(1) climb(2) climb(3) climb(4)`, the LLM often returns a strategy such as, `climb(1) climb(2) climb(3) climb(4) climb(5) climb(6) climb(7) climb(8) climb(9) climb(10)` as the best response. An interesting point to mention is that we observed strategies that followed different arithmetic or geometric patterns from the LLM, such as `climb(1) climb(2) climb(4) climb(8) climb(16)`,

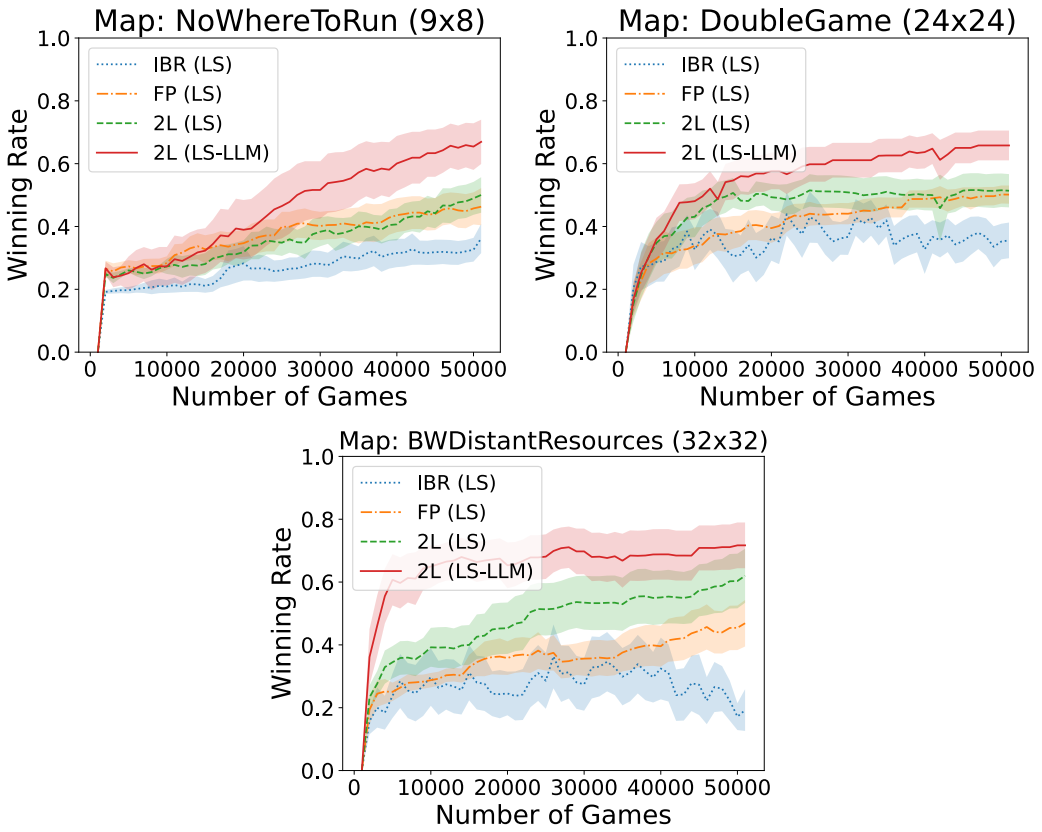


Figure 6.2: Average winning rate by the number of games played for LS-LLM, 2L, FP, and IBR in three maps of MicroRTS. The winning rate is computed by having the strategy a system synthesized, at a given number of games played, play against the strategies each of the other systems synthesized after the maximum number of games was played (50,000). The average and the 95% confidence interval are over 30 independent runs of the systems.

`climb(1)` `climb(3)` `climb(5)` `climb(7)` `climb(9)`, etc. This is as if the LLM were exploring the space of common sequences to see which one would represent a best response to the target strategy. For these types of responses, the "feedback-attempt" helped the LLM to gain insights about what went wrong and come up with a better strategy.

Figure 6.2 shows the results for MicroRTS, where the winning rate is computed by having the strategy a system generated, for a given number of games played, play against the strategy each of the other systems generated after the maximum number of games used in the experiment (50,000). LS-LLM outperforms all baselines by a large margin in the three maps. The programs it generates reach higher winning rates than the baselines. Especially for the

larger maps, 24×24 and 32×32, LS-LLM generates stronger strategies much more quickly than the baselines. For example, 2L has to play approximately 10 times more games than LS-LLM on the 32×32 map to reach the winning rate of approximately 60%.

Although the results on MicroRTS are similar to those in PR and CM, the explanation why LS-LLM performs better than the baselines is different in MicroRTS. The LLM is only able to directly compute best responses to σ_t in the early iterations of learning, when the strategies are weak. Later, as the system generates stronger strategies, the LLM fails to generate a best response. However, LS-LLM is more sample efficient in MicroRTS due to its initialization of the search in the neighborhood of a best response. Thus, the LLM helps the search explore different types of strategy. For example, the stronger strategies in the 9×8 map require the use of Ranged units. The LLM quickly generates programs that can reach game states whose features, such as Ranged units, would require the search alone many more iterations to reach. Often, the LLM generates a best response that incorporates a variety of combat units in response to a strategy that relies solely on workers for attacking the opponent. Moreover, it also focuses on returning strategies for defending the Base. For instance, it may suggest deploying ranged units to protect the Base, enabling attacks on the opponent before they come closer. These behaviors significantly accelerate the overall learning process.

6.2 Ablation Experiments

Figure 6.3 shows the results of the second (left plot) and third (right plot) experiments. 2L(LS-LLM-Enc) is the version of LS-LLM where we "encrypt" the name of the terminals of the DSL. In this experiment, 2L(LS-LLM-Enc) performs similarly to IBR and is worse than FP and 2L. We observed that the LLM struggles to generate strategies with encrypted symbols. The unsatisfactory winning rate indicates that it was unable to approximate a strategy that resides near the best response in the search space. At times, it produced flawed strategies, such as constructing a combat unit instead of training (units

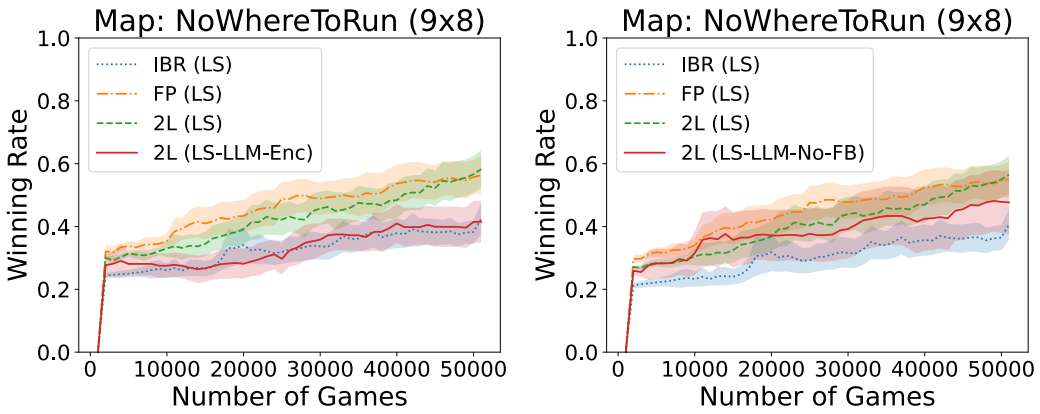


Figure 6.3: Average winning rate by the number of games played for 2L(LS-LLM-Enc) (left) and 2L(LS-LLM-No-FB) (right) in three maps of MicroRTS. The average and the 95% confidence interval are over 30 independent runs of the systems.

cannot be constructed, but only trained), which failed to meet the domain's constraints. These results show that much of the general knowledge that the LLM leverages to suggest initial candidates for search comes from the names of the functions and terminals.

2L(LS-LLM-No-FB) is the version of LS-LLM that does not provide feedback to the LLM. 2L(LS-LLM-No-FB) performs comparable to the baselines, suggesting that the feedback encodes valuable information. We conjecture that the feedback allows the LLM to verify whether the program it generates achieves the goals that the LLM sets to it. We make such a conjecture because the LLM often explains why it generated a strategy. For example, it could state that its strategy will train Ranged units to protect the Base. The feedback could show that no Ranged units are being trained because of not having enough resources, allowing the LLM to modify its strategy by prioritizing the harvest action to achieve its goal.

Chapter 7

Conclusion

In this dissertation, we presented LS-LLM, a novel approach designed to speed up local search for computing programmatic best responses in discontinuous programmatic search spaces, leveraging large language models (LLMs). Our approach involves retrieving the candidate strategy for the best response to a target strategy from the LLM. The LLM is used sparingly, only to initialize the search in the programmatic space, which makes LS-LLM a viable method given the often high monetary costs of using LLMs. We hypothesized that the use of an LLM would speed up the process of computing best responses in terms of the number of games that the system needs to play. We tested our hypothesis in three games that are challenging for current state-of-the-art systems, including a real-time strategy game. Our results supported our hypothesis, as the programmatic strategies LS-LLM generated were substantially stronger than those generated by current state-of-the-art systems.

7.1 Future Work

Future work involves investigating our approach in multi-player general-sum games, which would require additional computational effort to approximate the U value. Furthermore, potential connections to areas such as generalized planning and single-agent programmatic reinforcement learning can be explored. While we currently use LLMs to initialize the search, an extension could involve utilizing LLMs during the search for programmatic best responses. Enhancing the feedback on the LLM's previous attempt to find a

best response could also be beneficial. For instance, in microRTS, a textual summary of the last match between the last strategy to the target's support and the best response obtained from LLM could show more promising direction than the sample of action sequences. Additionally, an interesting direction for further research is to explore methods for selecting a subset of strategies to provide as input to the LLM, rather than only providing the last strategy to the target's support. One interesting area for future work could involve providing images of consecutive states following a sequence of actions, allowing the LLM to determine the reasons behind achieving those states due to the actions taken. Subsequently, the LLM could approximate the final state given a sequence of actions, and vice versa. Given that our approach was solely investigated with a closed-source LLM, conducting experiments with an open-source LLM and analyzing any potential data leakage that may arise could be considered as a potential future work.

References

Aguas, J. S., Jiménez, S., & Jonsson, A. (2018). Computing hierarchical finite state controllers with classical planning. *Journal of Artificial Intelligence Research*, *62*, 755–797. https://doi.org/10.1613/jair.1.11227

Aleixo, D. S., & Lelis, L. H. S. (2023). Show me the way! Bilevel search for synthesizing programmatic strategies. *Proceedings of the AAAI Conference on Artificial Intelligence*.

Alur, R., Bodik, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., & Udupa, A. (2013). Syntax-guided synthesis. *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design*, 1–17.

Ameen, S., & Lelis, L. H. S. (2023). Program synthesis with best-first bottom-up search. *Journal of Artificial Intelligence Research*, *77*, 1275–1310.

Balloccu, S., Schmidtová, P., Lango, M., & Dušek, O. (2024). Leak, cheat, repeat: Data contamination and evaluation malpractices in closed-source llms. *arXiv preprint arXiv:2402.03927*.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., & Tarlow, D. Deepcoder: Learning to write programs. In: In *Proceedings international conference on learning representations*. OpenReviews.net, 2017.

Barke, S., Peleg, H., & Polikarpova, N. (2020). Just-in-time learning for bottom-up enumerative synthesis. *Proceedings of the ACM on Programming Languages*, *4*(OOPSLA), 1–29.

Bastani, O., Pu, Y., & Solar-Lezama, A. (2018). Verifiable reinforcement learning via policy extraction. *Advances in Neural Information Processing Systems*, 2499–2509.

Bonet, B., Palacios, H., & Geffner, H. (2010). Automatic derivation of finite-state machines for behavior control. *Proceedings of the AAAI Conference on Artificial Intelligence*, 1656–1659.

Brown, G. (1951). Iterative solution of games by fictitious play. *Activity Analysis of Production and Allocation*, 374–376.

Butler, E., Torlak, E., & Popović, Z. (2017). Synthesizing interpretable strategies for solving puzzle games. *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 1–10.

Carvalho, T. H., Tjhia, K., & Lelis, L. (2024). Reclaiming the source of programmatic policies: Programmatic versus latent spaces. *The Twelfth In-*

ternational Conference on Learning Representations. https://openreview.net/forum?id=NGVljI6HkR

Celorrio, S. J., Aguas, J. S., & Jonsson, A. (2019). A review of generalized planning. *Knowl. Eng. Rev.*, *34*, e5. https://doi.org/10.1017/S0269888918000231

Chen, A., Dohan, D. M., & So, D. R. (2023). Evoprompting: Language models for code-level neural architecture search. *CoRR*, *abs/2302.14838*. https://doi.org/10.48550/ARXIV.2302.14838

Farzan, A., & Kincaid, Z. (2018). Strategy synthesis for linear arithmetic games. *Proc. ACM Program. Lang.*, *2*(POPL), 61:1–61:30. https://doi.org/10.1145/3158149

Gulwani, S. (2011). Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, *46*(1), 317–330.

Gulwani, S. (2016). Programming by examples (and its applications in data wrangling). *Verification and Synthesis of Correct and Secure Systems*.

Guo, P., Chen, Y., Tsai, Y., & Lin, S. (2023). Towards optimizing with large language models. *CoRR*, *abs/2310.05204*. https://doi.org/10.48550/ARXIV.2310.05204

Husien, I., & Schewe, S. (2016). Program generation using simulated annealing and model checking. *International Conference on Software Engineering and Formal Methods*, 155–171.

Inala, J. P., Bastani, O., Tavares, Z., & Solar-Lezama, A. (2020). Synthesizing programmatic policies that inductively generalize. *International Conference on Learning Representations*.

Jain, R., Ni, W., & Sunshine, J. (2023). Generating domain-specific programs for diagram authoring with large language models. *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 70–71.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by Simulated Annealing. *Science*, *220*(4598), 671–680. https://doi.org/10.1126/science.220.4598.671

Lanctot, M., Zambaldi, V., Gruslys, A., Lazaridou, A., Tuyls, K., Pérolat, J., Silver, D., & Graepel, T. (2017). A unified game-theoretic approach to multiagent reinforcement learning. *Advances in neural information processing systems*, *30*.

Lehman, J., Gordon, J., Jain, S., Ndousse, K., Yeh, C., & Stanley, K. O. (2022). Evolution through large models. *CoRR*, *abs/2206.08896*. https://doi.org/10.48550/ARXIV.2206.08896

Liang, J., Huang, W., Xia, F., Xu, P., Hausman, K., Ichter, B., Florence, P., & Zeng, A. (2023). Code as policies: Language model programs for embodied control. *IEEE International Conference on Robotics and Automation, ICRA 2023, London, UK, May 29 - June 2, 2023*, 9493–9500. https://doi.org/10.1109/ICRA48891.2023.10160591

Liu, F., Lin, X., Wang, Z., Yao, S., Tong, X., Yuan, M., & Zhang, Q. (2023). Large language model for multi-objective evolutionary optimization. *CoRR*, *abs/2310.12541*. https://doi.org/10.48550/ARXIV.2310.12541

Liu, S., Chen, C., Qu, X., Tang, K., & Ong, Y. (2023). Large language models as evolutionary optimizers. *CoRR*, *abs/2310.19046*. https://doi.org/10.48550/ARXIV.2310.19046

Liu, X., Chen, W., & Tan, M. (2023). Fidelity-induced interpretable policy extraction for reinforcement learning. *CoRR*, *abs/2309.06097*.

Mariño, J. R. H., Moraes, R. O., Oliveira, T. C., Toledo, C., & Lelis, L. H. S. (2021). Programmatic strategies for real-time strategy games. *Proceedings of the AAAI Conference on Artificial Intelligence*, *35*(1), 381–389.

Mariño, J. R. H., & Toledo, C. (2022). Evolving interpretable strategies for zero-sum games. *Applied Soft Computing*, 108860. https://doi.org/https://doi.org/10.1016/j.asoc.2022.108860

McMahan, H. B., Gordon, G. J., & Blum, A. (2003). Planning in the presence of cost functions controlled by an adversary. *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, 536–543.

Medeiros, L. C., Aleixo, D. S., & Lelis, L. H. S. (2022). What can we learn even from the weakest? Learning sketches for programmatic strategies. *Proceedings of the AAAI Conference on Artificial Intelligence*, 7761–7769.

Meyerson, E., Nelson, M. J., Bradley, H., Moradi, A., Hoover, A. K., & Lehman, J. (2023). Language model crossover: Variation through few-shot prompting. *CoRR*, *abs/2302.12170*. https://doi.org/10.48550/ARXIV.2302.12170

Milani, S., Zhang, Z., Topin, N., Shi, Z. R., Kamhoua, C. A., Papalexakis, E. E., & Fang, F. (2022). MAVIPER: learning decision tree policies for interpretable multi-agent reinforcement learning. *Machine Learning and Knowledge Discovery in Databases - European Conference*, *13716*, 251–266.

Moraes, R. O., Aleixo, D. S., Ferreira, L. N., & Lelis, L. H. S. (2023). Choosing well your opponents: How to guide the synthesis of programmatic strategies. *Proceedings of the International Joint Conference on Artificial Intelligence*, 4847–4854.

Nasir, M. U., Earle, S., Togelius, J., James, S., & Cleghorn, C. W. (2023). Llmatic: Neural architecture search via large language models and quality-diversity optimization. *CoRR*, *abs/2306.01102*. https://doi.org/10.48550/ARXIV.2306.01102

Odena, A., Shi, K., Bieber, D., Singh, R., Sutton, C., & Dai, H. (2021). BUSTLE: bottom-up program synthesis through learning-guided exploration. *International Conference on Learning Representations*.

Odena, A., & Sutton, C. (2020). Learning to represent programs with property signatures. *International Conference on Learning Representations*.

Ontañón, S. (2017). Combinatorial multi-armed bandits for real-time strategy games. *Journal of Artificial Intelligence Research*, *58*, 665–702.

Ontañón, S., Barriga, N. A., Silva, C. R., Moraes, R. O., & Lelis, L. H. S. (2018). The first microrts artificial intelligence competition. *AI Magazine*, *39*(1).

Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). Generative agents: Interactive simulacra of human behavior. *Proceedings of the Annual ACM Symposium on User Interface Software and Technology*, 2:1–2:22.

Qiu, W., & Zhu, H. (2022). Programmatic reinforcement learning without oracles. *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. https://openreview.net/forum?id=6Tk2noBdvxt

Shi, K., Dai, H., Ellis, K., & Sutton, C. (2022). Crossbeam: Learning to search in bottom-up program synthesis. *arXiv preprint arXiv:2203.10452*.

Silver, T., Dan, S., Srinivas, K., Tenenbaum, J. B., Kaelbling, L. P., & Katz, M. (2023). Generalized planning in PDDL domains with pretrained large language models. *CoRR*, *abs/2305.11014*. https://doi.org/10.48550/ARXIV.2305.11014

Singh, I., Blukis, V., Mousavian, A., Goyal, A., Xu, D., Tremblay, J., Fox, D., Thomason, J., & Garg, A. (2023). Progprompt: Generating situated robot task plans using large language models. *IEEE International Conference on Robotics and Automation, ICRA 2023, London, UK, May 29 - June 2, 2023*, 11523–11530. https://doi.org/10.1109/ICRA48891.2023.10161317

Solar-Lezama, A. (2009). The sketching approach to program synthesis. *Asian Symposium on Programming Languages and Systems*, 4–13.

Trivedi, D., Zhang, J., Sun, S., & Lim, J. J. (2021). Learning to synthesize programs as interpretable and generalizable policies. *Advances in Neural Information Processing Systems*, 25146–25163.

Verma, A., Murali, V., Singh, R., Kohli, P., & Chaudhuri, S. (2018a). Programmatically interpretable reinforcement learning. *Proceedings of the International Conference on Machine Learning*, 5052–5061.

Verma, A., Murali, V., Singh, R., Kohli, P., & Chaudhuri, S. (2018b, October). Programmatically interpretable reinforcement learning. In J. Dy & A. Krause (Eds.), *Proceedings of the 35th international conference on machine learning* (pp. 5045–5054, Vol. 80). PMLR. https://proceedings.mlr.press/v80/verma18a.html

Wang, B., Wang, Z., Wang, X., Cao, Y., A. Saurous, R., & Kim, Y. (2023). Grammar prompting for domain-specific language generation with large language models. In A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, & S. Levine (Eds.), *Advances in neural information processing systems* (pp. 65030–65055, Vol. 36). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2023/file/cd40d0d65bfebb894ccc9ea822b47fa8-Paper-Conference.pdf

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., & Anandkumar, A. (2023). Voyager: An open-ended embodied agent with

large language models. *CoRR*, *abs/2305.16291*. https://doi.org/10.48550/ARXIV.2305.16291

Wang, Z., Cai, S., Liu, A., Ma, X., & Liang, Y. (2023). Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. *CoRR*, *abs/2302.01560*. https://doi.org/10.48550/ARXIV.2302.01560

Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q. V., Zhou, D., & Chen, X. (2023). Large language models as optimizers. *CoRR*, *abs/2309.03409*. https://doi.org/10.48550/ARXIV.2309.03409

Zhao, Z., Lee, W. S., & Hsu, D. (2023). Large language models as commonsense knowledge for large-scale task planning. *CoRR*, *abs/2305.14078*. https://doi.org/10.48550/ARXIV.2305.14078

Zhu, X., Chen, Y., Tian, H., Tao, C., Su, W., Yang, C., Huang, G., Li, B., Lu, L., Wang, X., Qiao, Y., Zhang, Z., & Dai, J. (2023). Ghost in the minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory. *CoRR*, *abs/2305.17144*. https://doi.org/10.48550/ARXIV.2305.17144

Appendix A

Supplementary Materials

In this document, we provide the maps used in the MicroRTS experiments, and the prompts used in all three domains. The domain-specific languages (DSLs) used in the experiments are presented in the prompts. Section A.2.4 shows the encrypted domain-specific language used for one of the ablation experiments. In Sections A.2.1, A.2.2 and A.2.3, we present descriptions of the three maps within the prompts used.

Note that the First Attempt and Feedback Attempt prompts for MicroRTS also present examples of strategies written in the Microlanguage.

A.1 MicroRTS Maps

Figures A.1, A.2, and A.3 show the three MicroRTS maps used in our experiments.

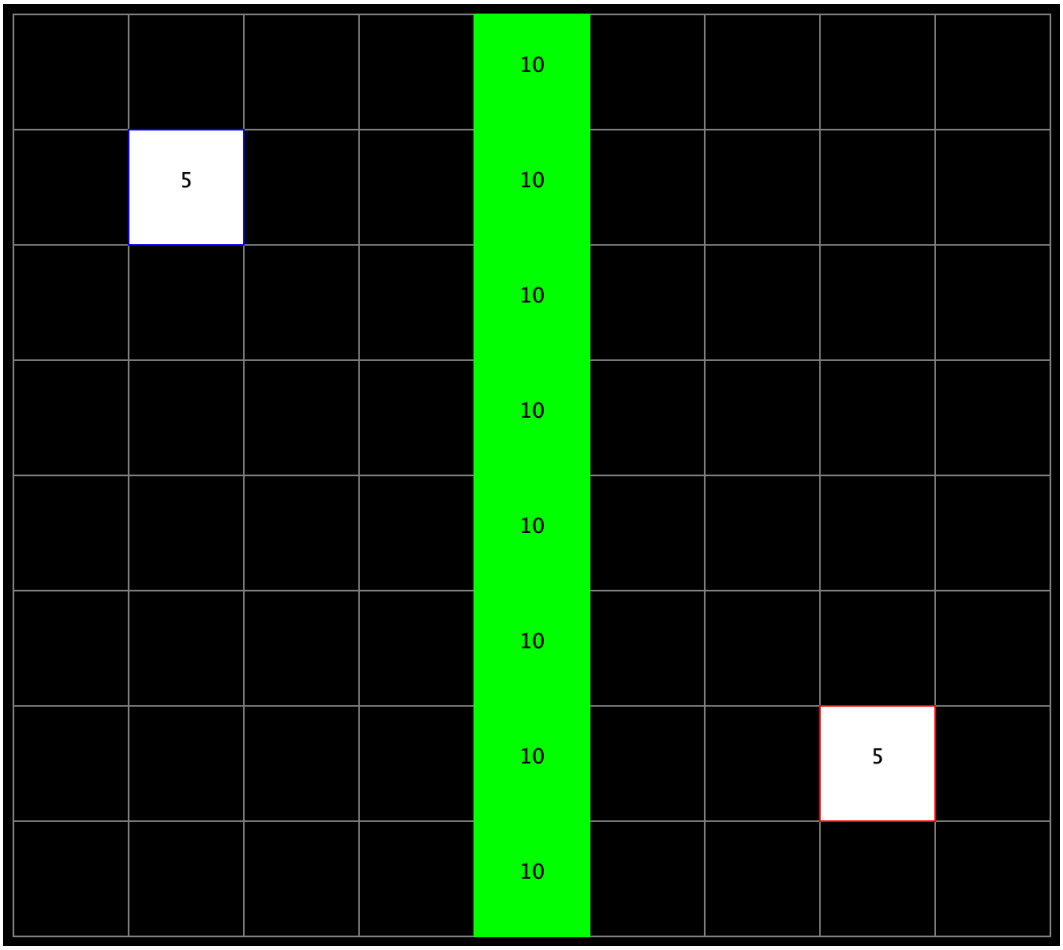


Figure A.1: NoWhereToRun (9x8)

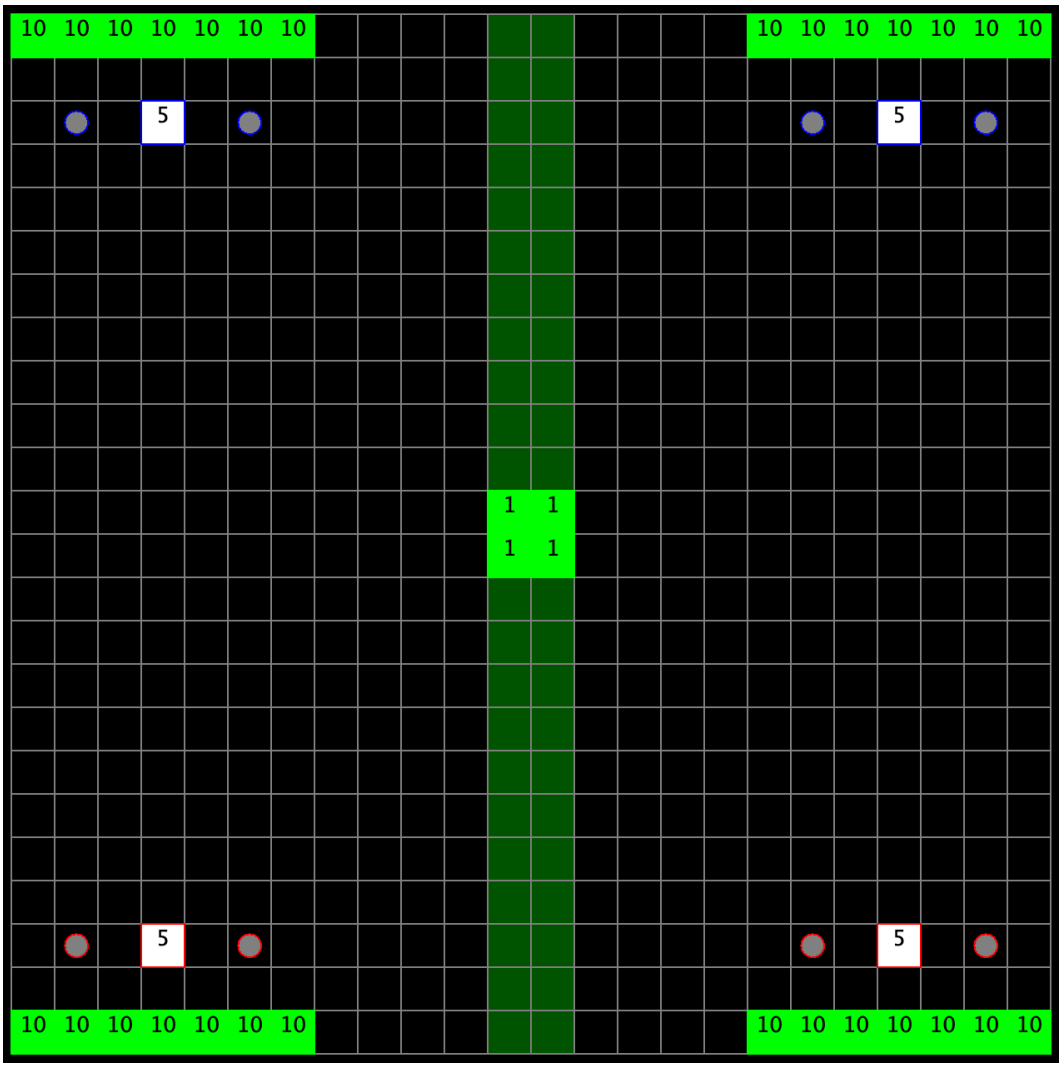


Figure A.2: DoubleGame (24x24)

A.2 MicroRTS Prompts

A.2.1 Initial Attempt (Example for 9x8 Map)

Consider a 9x8 gridded map of microRTS, a real-time strategy game. Consider this map as a 2 dimensional array with the following structure:

- There are a total of 8 neutral resource cells situated along the central column of the map, dividing the map into two parts. Each resource cell contains 10 units of resources.
- The base B1 of player 1 is located at index (1,1), which is located on the left side of the map.
- The base B2 of player 2 is located at index (7,6), which is located on

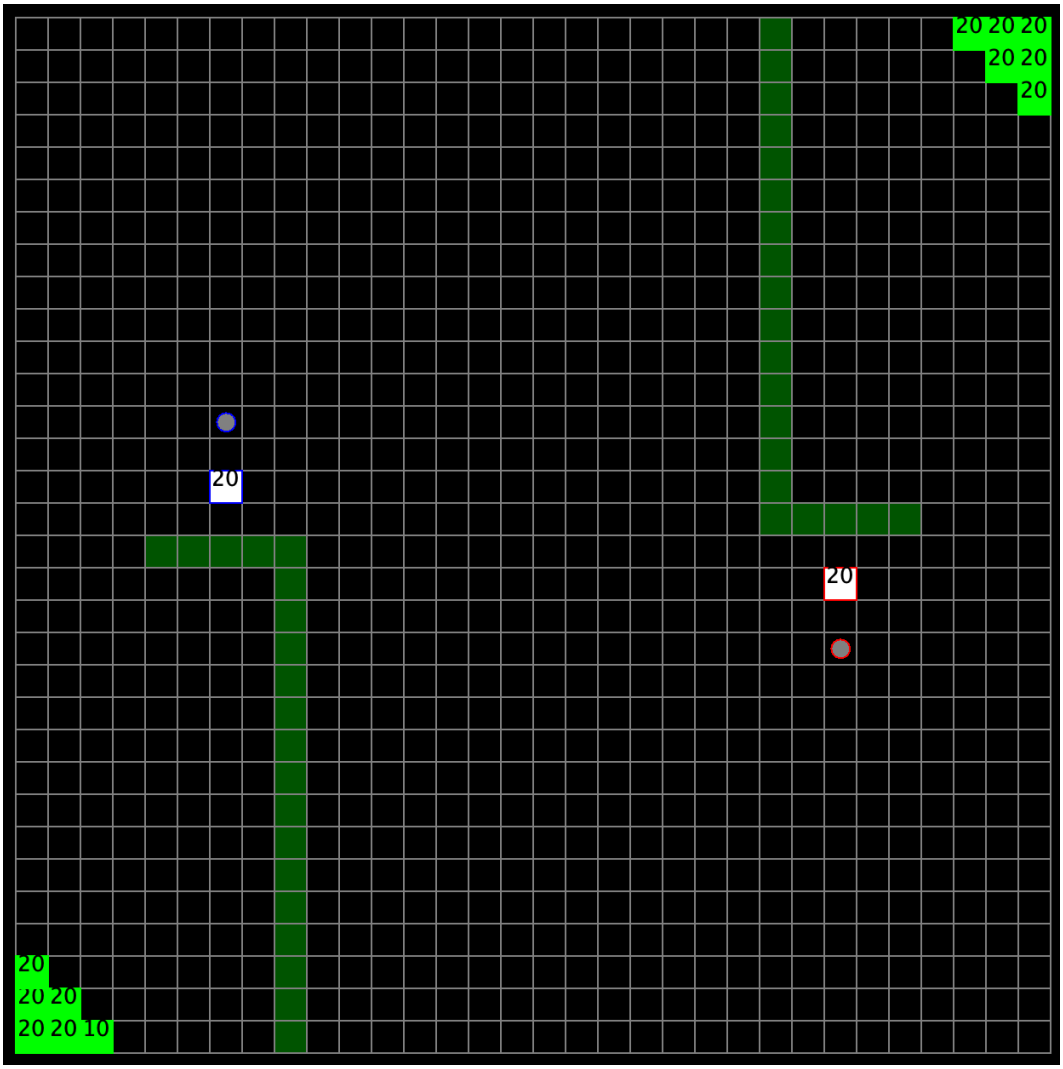


Figure A.3: BWDistantResources (32x32)

the right side of the map.

– Each player controls one base, which initially has 5 units of resources.

– The only unit a player controls at the beginning of the game is the base.

Consider this Context-Free Grammar (CFG) describing a programming language for writing programs encoding strategies of microRTS. The CFG is shown in the $< CFG >< /CFG >$ tag bellow:

$< CFG >$

$S \rightarrow SS$ | for(Unit u) S | if(B) then S
| if(B) then S else S | C | λ

$B \rightarrow u$.hasNumberOfUnits(T, N)
| u.opponentHasNumberOfUnits(T, N)
| u.hasLessNumberOfUnits(T, N)
| u.haveQtdUnitsAttacking(N)
| u.hasUnitWithinDistanceFromOpponent(N)
| u.hasNumberOfWorkersHarvesting(N)
| u.is_Type(T)
| u.isBuilder()
| u.canAttack()
| u.hasUnitThatKillsInOneAttack()
| u.opponentHasUnitThatKillsUnitInOneAttack()
| u.hasUnitInOpponentRange()
| u.opponentHasUnitInPlayerRange()
| u.canHarvest()

$C \rightarrow u$.build(T, D, N) | u.train(T, D, N) | u.moveToUnit(T_p, O_p)
| u.attack(O_p) | u.harvest(N)
| u.attackIfInRange() | u.moveAway()

$T \rightarrow$ Base | Barracks | Ranged | Heavy
| Light | Worker

$N \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
| 10 | 15 | 20 | 25 | 50 | 100

$D \rightarrow$ EnemyDir | Up | Down | Right | Left

$O_p \rightarrow$ Strongest | Weakest | Closest | Farthest
| LessHealthy | MostHealthy | Random

$T_p \rightarrow$ Ally | Enemy

$< /CFG >$

This language allows nested loops and conditionals. It contains several Boolean functions (B) and command-oriented functions (C) that provide either information about the current state of the game or commands for the ally units.
The Boolean functions ('B' in the CFG) are described below:

1. u.hasNumberOfUnits(T, N): Checks if the ally player has N units of type T.

2. u.opponentHasNumberOfUnits(T, N): Checks if the opponent player has N units of type T.

3. u.hasLessNumberOfUnits(T, N): Checks if the ally player has less than N units of type T.

4. u.haveQtdUnitsAttacking(N): Checks if the ally player has N units attacking the opponent.

5. u.hasUnitWithinDistanceFromOpponent(N): Checks if the ally player has a unit within a distance N from a opponent's unit.

6. u.hasNumberOfWorkersHarvesting(N): Checks if the ally player has N units of type Worker harvesting resources.

7. u.is_Type(T): Checks if a unit is an instance of type T.

8. u.isBuilder(): Checks if a unit is of type Worker.

9. u.canAttack(): Checks if a unit can attack.

10. u.hasUnitThatKillsInOneAttack(): Checks if the ally player has a unit that kills an opponent's unit with one attack action.

11. u.opponentHasUnitThatKillsUnitInOneAttack(): Checks if the opponent player has a unit that kills an ally's unit with one attack action.

12. u.hasUnitInOpponentRange(): Checks if an unit of the ally player is within attack range of an opponent's unit.

13. u.opponentHasUnitInPlayerRange(): Checks if an unit of the opponent player is within attack range of an ally's unit.

14. u.canHarvest(): Checks if a unit can harvest resources.

The Command functions ('C' in the CFG) are described below:

1. u.build(T, D, N): Builds N units of type T on a cell located on the D direction of the unit.

2. u.train(T, D, N): Trains N units of type T on a cell located on the D direction of the structure responsible for training them.

3. u.moveToUnit(T_p, O_p): Commands a unit to move towards the player T_p following a criterion O_p.

4. u.attack(O_p): Sends N Worker units to harvest resources.

5. u.harvest(N): Sends N Worker units to harvest resources.

6. u.attackIfInRange(): Commands a unit to stay idle and attack if an opponent unit comes within its attack range.

7. u.moveAway(): Commands a unit to move in the opposite direction of the player's base.

'T' represents the types a unit can assume. 'N' is a set of integers. 'D' represents the directions available used in action functions.
'O_p' is a set of criteria to select an opponent unit based on their current state. 'T_p' represents the set of target players.

The following 5 are some guidelines for writing the playing strategy:

1. There is NO NEED TO write classes or initiate objects such as Unit, Worker, etc. There is also NO NEED TO write comments.

2. Use curly braces like C/C++/Java while writing any 'for' or 'if' or 'if-else' block. Start the curly braces in the same line of the block.

3. Do not write 'else if(B) {' block. Write 'else { if(B) {...}}' instead.

4. A strategy must be written inside one or multiple 'for' blocks.

5. You must not use any symbols (for example: &&, ||, etc.) outside the CFG. In case of code like 'if (B1 && B2)', write 'if (B1) { if (B2) {...}}' instead.

Now your tasks are the following 7:

1. Understand the Boolean (B) and command (C) functions from above and try to relate them in the context of microRTS playing strategies.

2. Write a program in the microRTS language encoding a very strong game-playing strategy for the 9x8 map described above. You must follow the guidelines for writing the playing strategy while writing your program.

3. You must not use any symbols (for example &&, ||, etc.) that the CFG does not accept. You have to strictly follow the CFG while writing the program.

4. Look carefully, the methods of non-terminal symbols B and C have prefixes 'u.' in the examples since they are methods of the object 'Unit u'. You should follow the patterns of the examples.

5. Write only the pseudocode inside '$< strategy >< /strategy >$' tag.

6. Do not write unnecessary symbols of the CFG such as, '$S \rightarrow$', '$\rightarrow$', etc.

7. Check the program and ensure it does not violate the rules of the CFG or the guidelines for writing the strategy.

A.2.2 First Attempt (Example for 24x24 Map)

Consider a 24x24 gridded map of microRTS, a real-time strategy game. Consider this map as a 2 dimensional array with the following structure:

- There is a wall in the middle of the map consisting of two columns that has a small passage of 4 cells. The small passage consists of 4 resource cells each having only 1 resource.
- There are 28 resource cells at the top-left, top-right, bottom-left and bottom-right corners of the map respectively where each of them contains 10 units of resources.
- The bases of player 1 are located at indices (3,2) and (20,2), located on both sides of the wall.
- The bases of player 2 are located at indices (20,21) and (3,21), also located on both sides of the wall.
- Each player controls two bases, which initially have 5 units of resources each.
- There are 2 workers beside each base. So a total of 4 workers for each of the players.

Consider this Context-Free Grammar (CFG) describing a programming language for writing programs encoding strategies of microRTS. The CFG is shown in the $< CFG >< /CFG >$ tag bellow:

$< CFG >$

$S \rightarrow SS$ | for(Unit u) S | if(B) then S
| if(B) then S else S | C | λ

$B \rightarrow u$.hasNumberOfUnits(T, N)
| u.opponentHasNumberOfUnits(T, N)
| u.hasLessNumberOfUnits(T, N)
| u.haveQtdUnitsAttacking(N)
| u.hasUnitWithinDistanceFromOpponent(N)
| u.hasNumberOfWorkersHarvesting(N)
| u.is_Type(T)
| u.isBuilder()
| u.canAttack()
| u.hasUnitThatKillsInOneAttack()
| u.opponentHasUnitThatKillsUnitInOneAttack()
| u.hasUnitInOpponentRange()
| u.opponentHasUnitInPlayerRange()
| u.canHarvest()

$C \rightarrow u$.build(T, D, N) | u.train(T, D, N) | u.moveToUnit(T_p, O_p)
| u.attack(O_p) | u.harvest(N)
| u.attackIfInRange() | u.moveAway()

$T \rightarrow$ Base | Barracks | Ranged | Heavy
| Light | Worker

$N \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
| 10 | 15 | 20 | 25 | 50 | 100

$D \rightarrow$ EnemyDir | Up | Down | Right | Left

$O_p \rightarrow$ Strongest | Weakest | Closest | Farthest
| LessHealthy | MostHealthy | Random

$T_p \rightarrow$ Ally | Enemy

$< /CFG >$

This language allows nested loops and conditionals. It contains several Boolean functions (B) and command-oriented functions (C) that provide either information about the current state of the game or commands for the ally units.
The Boolean functions ('B' in the CFG) are described below:

1. u.hasNumberOfUnits(T, N): Checks if the ally player has N units of type T.

2. u.opponentHasNumberOfUnits(T, N): Checks if the opponent player has N units of type T.

3. u.hasLessNumberOfUnits(T, N): Checks if the ally player has less than N units of type T.

4. u.haveQtdUnitsAttacking(N): Checks if the ally player has N units attacking the opponent.

5. u.hasUnitWithinDistanceFromOpponent(N): Checks if the ally player has a unit within a distance N from a opponent's unit.

6. u.hasNumberOfWorkersHarvesting(N): Checks if the ally player has N units of type Worker harvesting resources.

7. u.is_Type(T): Checks if a unit is an instance of type T.

8. u.isBuilder(): Checks if a unit is of type Worker.

9. u.canAttack(): Checks if a unit can attack.

10. u.hasUnitThatKillsInOneAttack(): Checks if the ally player has a unit that kills an opponent's unit with one attack action.

11. u.opponentHasUnitThatKillsUnitInOneAttack(): Checks if the opponent player has a unit that kills an ally's unit with one attack action.

12. u.hasUnitInOpponentRange(): Checks if an unit of the ally player is within attack range of an opponent's unit.

13. u.opponentHasUnitInPlayerRange(): Checks if an unit of the opponent player is within attack range of an ally's unit.

14. u.canHarvest(): Checks if a unit can harvest resources.

The Command functions ('C' in the CFG) are described below:

1. u.build(T, D, N): Builds N units of type T on a cell located on the D direction of the unit.

2. u.train(T, D, N): Trains N units of type T on a cell located on the D direction of the structure responsible for training them.

3. u.moveToUnit(T_p, O_p): Commands a unit to move towards the player T_p following a criterion O_p.

4. u.attack(O_p): Sends N Worker units to harvest resources.

5. u.harvest(N): Sends N Worker units to harvest resources.

6. u.attackIfInRange(): Commands a unit to stay idle and attack if an opponent unit comes within its attack range.

7. u.moveAway(): Commands a unit to move in the opposite direction of the player's base.

'T' represents the types a unit can assume. 'N' is a set of integers. 'D' represents the directions available used in action functions.
'O_p' is a set of criteria to select an opponent unit based on their current state. 'T_p' represents the set of target players.

Now consider the following program encoding a strategy for playing microRTS written inside '$< strategy-1 >< /strategy-1 >$' tag:
$< strategy-1 >$

```
for(Unit u){
  for(Unit u){
    u.build(Barracks, Right, 50)
  }
  u.train(Worker, Up, 4)
  u.attack(Strongest)
  for(Unit u){
    u.harvest(5)
  }
  u.moveToUnit(Ally, Closest)
  for(Unit u){
    u.train(Ranged, Up, 5)
  }
}
```

$< /strategy-1 >$

Now your tasks are the following 3:

1. Analyze strategy-1 and try to analyze its weaknesses.

2. Write a new strategy that defeats strategy-1.

3. You need to only write this new strategy inside '$< counterStrategy >< /counterStrategy >$' tag.

A.2.3 Feedback Attempt (Example for 32x32 Map)

Consider a 32x32 map of microRTS, a real-time strategy game. Consider this map as a 2 dimensional array with the following structure:

- There are two L-shaped obstacles on the map, each with a passage of 4 cells located at the middle of left and right sides.
- There are a total of 12 neutral resource cells R located at the top-right and bottom-left corners of the map. Each resource center contains 20 units of resources.
- The base B1 of player 1 is located at index (6,14), which is located on the left side of the map.
- The base B2 of player 2 is located at index (25,17), which is located on the right side of the map.
- Each player controls one Base, which initially has 20 units of resources.
- There is one worker for each player besides their bases.

Consider this Context-Free Grammar (CFG) describing a programming language for writing programs encoding strategies of microRTS. The CFG is shown in the $< CFG >< /CFG >$ tag bellow:

$< CFG >$

$S \rightarrow SS$ | for(Unit u) S | if(B) then S
| if(B) then S else S | C | λ

$B \rightarrow u$.hasNumberOfUnits(T, N)
| u.opponentHasNumberOfUnits(T, N)
| u.hasLessNumberOfUnits(T, N)
| u.haveQtdUnitsAttacking(N)
| u.hasUnitWithinDistanceFromOpponent(N)
| u.hasNumberOfWorkersHarvesting(N)
| u.is_Type(T)
| u.isBuilder()
| u.canAttack()
| u.hasUnitThatKillsInOneAttack()
| u.opponentHasUnitThatKillsUnitInOneAttack()
| u.hasUnitInOpponentRange()
| u.opponentHasUnitInPlayerRange()
| u.canHarvest()

$C \rightarrow u$.build(T, D, N) | u.train(T, D, N) | u.moveToUnit(T_p, O_p)
| u.attack(O_p) | u.harvest(N)
| u.attackIfInRange() | u.moveAway()

$T \rightarrow$ Base | Barracks | Ranged | Heavy
| Light | Worker

$N \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
| 10 | 15 | 20 | 25 | 50 | 100

$D \rightarrow$ EnemyDir | Up | Down | Right | Left

$O_p \rightarrow$ Strongest | Weakest | Closest | Farthest
| LessHealthy | MostHealthy | Random

$T_p \rightarrow$ Ally | Enemy

$< /CFG >$

This language allows nested loops and conditionals. It contains several Boolean functions (B) and command-oriented functions (C) that provide either information about the current state of the game or commands for the ally units.
The Boolean functions ('B' in the CFG) are described below:

1. u.hasNumberOfUnits(T, N): Checks if the ally player has N units of type T.

2. u.opponentHasNumberOfUnits(T, N): Checks if the opponent player has N units of type T.

3. u.hasLessNumberOfUnits(T, N): Checks if the ally player has less than N units of type T.

4. u.haveQtdUnitsAttacking(N): Checks if the ally player has N units attacking the opponent.

5. u.hasUnitWithinDistanceFromOpponent(N): Checks if the ally player has a unit within a distance N from a opponent's unit.

6. u.hasNumberOfWorkersHarvesting(N): Checks if the ally player has N units of type Worker harvesting resources.

7. u.is_Type(T): Checks if a unit is an instance of type T.

8. u.isBuilder(): Checks if a unit is of type Worker.

9. u.canAttack(): Checks if a unit can attack.

10. u.hasUnitThatKillsInOneAttack(): Checks if the ally player has a unit that kills an opponent's unit with one attack action.

11. u.opponentHasUnitThatKillsUnitInOneAttack(): Checks if the opponent player has a unit that kills an ally's unit with one attack action.

12. u.hasUnitInOpponentRange(): Checks if an unit of the ally player is within attack range of an opponent's unit.

13. u.opponentHasUnitInPlayerRange(): Checks if an unit of the opponent player is within attack range of an ally's unit.

14. u.canHarvest(): Checks if a unit can harvest resources.

The Command functions ('C' in the CFG) are described below:

1. u.build(T, D, N): Builds N units of type T on a cell located on the D direction of the unit.

2. u.train(T, D, N): Trains N units of type T on a cell located on the D direction of the structure responsible for training them.

3. u.moveToUnit(T_p, O_p): Commands a unit to move towards the player T_p following a criterion O_p.

4. u.attack(O_p): Sends N Worker units to harvest resources.

5. u.harvest(N): Sends N Worker units to harvest resources.

6. u.attackIfInRange(): Commands a unit to stay idle and attack if an opponent unit comes within its attack range.

7. u.moveAway(): Commands a unit to move in the opposite direction of the player's base.

'T' represents the types a unit can assume. 'N' is a set of integers. 'D' represents the directions available used in action functions.
'O_p' is a set of criteria to select an opponent unit based on their current state. 'T_p' represents the set of target players.

Now consider the following program encoding a strategy for playing microRTS written inside '$< strategy - 1 >< /strategy - 1 >$' tag:
$< strategy - 1 >$

```
for(Unit u){
  for(Unit u){
    u.build(Barracks, Right, 50)
  }
  u.train(Worker, Up, 4)
  u.attack(Strongest)
  for(Unit u){
    u.harvest(5)
  }
  u.moveToUnit(Ally, Closest)
  for(Unit u){
    u.train(Ranged, Up, 5)
  }
}
```

$< /strategy - 1 >$

Here is a strategy that could not defeat the above strategy:
$< strategy - 2 >$

```
for(Unit u){
  for(Unit u){
    u.build(Barracks,Left,1)
  }
  u.train(Heavy,Down,4)
  u.train(Worker,Up,4)
  u.attack(Strongest)
  for(Unit u){
    u.harvest(5)
  }
  u.moveToUnit(Enemy,Closest)
```

```
  for(Unit u){
    u.train(Ranged,Up,5)
  }
}
```

$< /strategy - 2 >$

The following is an encoding of the units, which we will use to give you information about a match played between strategy-1 and strategy-2 above.
Base : B
Worker : W
Ranged : Rg
Light : Li
Heavy : Hv
Barracks : Br

The following is an encoding of the actions:
attack_location : att_loc
return : ret
wait : wt
move : mv
produce : prod
harvest : har

The following is an encoding of the directions:
left : l
right : r
up : u
down : d

The following is a randomly sampled sequence of actions of the match played between strategy-1 as player 0 and strategy-2 as player 1:

```
 { (B(4)(0, (1,1), 10, 0),prod(u,W)) },
{ (B(6)(1, (7,6), 10, 0),prod(u,W))(W(16)(1, (6,5), 1,
1),mv(u))(W(18)(1, (6,6), 1, 0),mv(l)) },
{ (W(17)(0, (0,3), 1, 1),mv(u))(W(19)(0, (1,0), 1,
0),mv(r))(W(21)(0, (2,1), 1, 1),ret(l))(B(4)(0, (1,1), 10,
0),wt(10)) },
{ (W(15)(0, (1,2), 1, 0),mv(d))(W(19)(0, (3,0), 1,
1),mv(l))(W(21)(0, (3,1), 1, 1),mv(l))(B(4)(0, (1,1), 10,
0),wt(10))(W(17)(0, (0,1), 1, 0),wt(10))(Br(23)(0, (2,2), 4,
0),wt(10)) },
{ (W(18)(1, (7,7), 1, 0),mv(l))(W(22)(1, (5,6), 1,
1),mv(r))(Rg(26)(1, (5,4), 1, 0),mv(u))(B(6)(1, (7,6), 10,
```

```
0),wt(10))(W(16)(1, (6,5), 1, 0),wt(10))(W(20)(1, (7,5), 1,
0),wt(10))(Br(24)(1, (5,5), 4, 0),wt(10)) },
{ (Rg(25)(0, (3,2), 1, 0),wt(10)) },
{ (Rg(28)(0, (2,5), 1, 0),att_loc(5,5)) },
{ (B(6)(1, (7,6), 10, 0),prod(l,W))(W(33)(1, (7,5), 1,
0),mv(u))(W(37)(1, (7,7), 1, 0),mv(l))(Br(36)(1, (6,5), 4,
0),wt(10)) },
{ (Rg(25)(0, (5,4), 1, 0),att_loc(7,6))(Rg(31)(0, (6,4), 1,
0),att_loc(7,6))(Rg(28)(0, (3,6), 1, 0),wt(10)) },
{ (B(6)(1, (7,6), 2, 0),wt(10)) }
```
The strategy-2 failed to defeat strategy-1.

Now your tasks are the following 3:

1. Analyze strategy-1 and try to analyze its weaknesses. For this analysis, you may take help from the sequence of actions from the match between strategy-1 and strategy-2 we provided.
2. Write a new strategy that defeats strategy-1.
3. You need to only write this new strategy inside '$< counterStrategy >< /counterStrategy >$' tag.

A.2.4 Encrypted DSL

Consider this Context-Free Grammar (CFG) describing a programming language for writing programs encoding strategies of microRTS. The CFG is shown in the $< CFG >< /CFG >$ tag bellow:

$< CFG >$

$S \rightarrow SS$ | for(Unit u) S | if(B) then S
| if(B) then S else S | C | λ

$B \rightarrow u.\text{b1}(T, N)$
| $u.\text{b2}(T, N)$
| $u.\text{b3}(T, N)$
| $u.\text{b4}(N)$
| $u.\text{b5}(N)$
| $u.\text{b6}(N)$
| $u.\text{b7}(\text{T})$
| $u.\text{b8}()$
| $u.\text{b9}()$
| $u.\text{b10}()$
| $u.\text{b11}()$
| $u.\text{b12}()$
| $u.\text{b13}()$
| $u.\text{b14}()$

$C \rightarrow u.\text{c1}(T, D, N) \mid u.\text{c2}(T, D, N) \mid u.\text{c3}(T_p, O_p)$
| $u.\text{c4}(O_p) \mid u.\text{c5}(N)$
| $u.\text{c6}() \mid u.\text{c7}()$

$T \rightarrow \text{t1} \mid \text{t2} \mid \text{t3} \mid \text{t4}$
| t5 | t6

$N \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
| 10 | 15 | 20 | 25 | 50 | 100

$D \rightarrow \text{d1} \mid \text{d2} \mid \text{d3} \mid \text{d4} \mid \text{d5}$

$O_p \rightarrow \text{op1} \mid \text{op2} \mid \text{op3} \mid \text{op4}$
| op5 | op6 | op7

$T_p \rightarrow \text{tp1} \mid \text{tp2}$

$< /CFG >$

This language allows nested loops and conditionals. It contains several Boolean functions (B) and command-oriented functions (C) that provide either information about the current state of the game or commands for the ally units.
The Boolean functions ('B' in the CFG) are described below:

1. u.b1(T, N): Checks if the ally player has N units of type T.

2. u.b2(T, N): Checks if the opponent player has N units of type T.
3. u.b3(T, N): Checks if the ally player has less than N units of type T.
4. u.b4(N): Checks if the ally player has N units attacking the opponent.
5. u.b5(N): Checks if the ally player has a unit within a distance N from a opponent's unit.
6. u.b6(N): Checks if the ally player has N units of type Worker harvesting resources.
7. u.b7(T): Checks if a unit is an instance of type T.
8. u.b8(): Checks if a unit is of type Worker.
9. u.b9(): Checks if a unit can attack.
10. u.b10(): Checks if the ally player has a unit that kills an opponent's unit with one attack action.
11. u.b11(): Checks if the opponent player has a unit that kills an ally's unit with one attack action.
12. u.b12(): Checks if an unit of the ally player is within attack range of an opponent's unit.
13. u.b13(): Checks if an unit of the opponent player is within attack range of an ally's unit.
14. u.b14(): Checks if a unit can harvest resources.

The Command functions ('C' in the CFG) are described below:

1. u.c1(T, D, N): Builds N units of type T on a cell located on the D direction of the unit.
2. u.c2(T, D, N): Trains N units of type T on a cell located on the D direction of the structure responsible for training them.
3. u.c3(T_p, O_p): Commands a unit to move towards the player T_p following a criterion O_p.
4. u.c4(O_p): Sends N Worker units to harvest resources.
5. u.c5(N): Sends N Worker units to harvest resources.
6. u.c6(): Commands a unit to stay idle and attack if an opponent unit comes within its attack range.

7. u.c7(): Commands a unit to move in the opposite direction of the player's base.

'T' represents the types of units as the following:

1. t1: Base
2. t2: Barracks
3. t3: Ranged
4. t4: Heavy
5. t5: Light
6. t6: Worker

'D' represents directions as the following:

1. d1: EnemyDir
2. d2: Up
3. d3: Down
4. d4: Right
5. d5: Left

'O_p' is a set of criteria to select an opponent unit based on its current state like the following:

1. op1: Strongest
2. op2: Weakest
3. op3: Closest
4. op4: Farthest
5. op5: LessHealthy
6. op6: MostHealthy

'T_p' represents the set of target players like the following:

1. tp1: Ally
2. tp2: Enemy

Finally, 'N' is a set of integers.

A.3 Poachers and Rangers Prompts

A.3.1 Initial Attempt

We have an environment called 'Poachers and Rangers' where 2 teams called poachers and rangers are competing with each other in a national park and its surroundings. The park has 60 gates in total. The goal for each team is to defeat the opponents.

Now I have the following CFG to write programs for poachers in the above environment:
CFG for Poachers:

$$S \rightarrow SA \mid A$$
$$A \rightarrow attack(n)$$
$$n \rightarrow 1 \mid 2 \mid 3 \mid \ldots \mid 59 \mid 60$$

The following is the CFG to write programs for rangers:
CFG for Rangers:

$$S \rightarrow SA \mid A$$
$$A \rightarrow defend(n)$$
$$n \rightarrow 1 \mid 2 \mid 3 \mid \ldots \mid 59 \mid 60$$

The following is the explanation of the above CFG:
CFG Explanation:

S: Starting symbol that can contain one or multiple actions.

A: Refers to the action taken by the team.

attack(n): Refers to the action to attack the n-th gate of the park

defend(n): Refers to the action to defend the n-th gate of the park

n: Any positive integer up to 60.

. . . : It is not part of the CFG. It has been used to indicate all positive numbers in between.

The following are some guidelines for writing the strategy program:
Strategy Writing Guidelines:

1. There is NO NEED TO write classes or initiate objects such as Poachers, Rangers, Gates, etc. There is NO NEED TO write comments.

2. DO NOT write '. . . ' in the program, since it is not a part of the CFG.

3. Write only the action or the sequence of actions such as 'attack(a)' or 'attack(a) attack(b) attack(c)' where a, b and c are positive integers.

Now your tasks are the following 5:

1. Understand all the symbols of the given CFG in the context of this given 'Poachers and Rangers' environment.

2. Write a very strong strategy for poachers/rangers considering the given environment, the above CFG and the strategy writing guidelines.

3. You must not use any symbols (for example '$S \rightarrow$', '$\rightarrow$', '|', etc.) outside the given CFG. Write only the action or the sequence of action as mentioned in the strategy writing guideline.

4. Replace '...' with contents meant by this symbol if there are any, then write only the strategy program inside '$< strategy >< /strategy >$' tag.

5. Check the strategy program and ensure it does not violate the rules of the CFG or the guidelines for writing the strategy.

A.3.2 First Attempt

We have an environment called 'Poachers and Rangers' where 2 teams called poachers and rangers are competing with each other in a national park and its surroundings. The park has 60 gates in total. The goal for each team is to defeat the opponents.

Now I have the following CFG to write programs for poachers in the above environment:
CFG for Poachers:

$$S \rightarrow SA \mid A$$
$$A \rightarrow attack(n)$$
$$n \rightarrow 1 \mid 2 \mid 3 \mid \ldots \mid 59 \mid 60$$

The following is the CFG to write programs for rangers:

CFG for Rangers:

$$S \rightarrow SA \mid A$$
$$A \rightarrow defend(n)$$
$$n \rightarrow 1 \mid 2 \mid 3 \mid \ldots \mid 59 \mid 60$$

The following is the explanation of the above CFG:
CFG Explanation:

S: Starting symbol that can contain one or multiple actions.

A: Refers to the action taken by the team.

attack(n): Refers to the action to attack the n-th gate of the park

defend(n): Refers to the action to defend the n-th gate of the park

n: Any positive integer up to 60.

. . . : It is not part of the CFG. It has been used to indicate all positive numbers in between.

The following are some guidelines for writing the strategy program:
Strategy Writing Guidelines:

1. There is NO NEED TO write classes or initiate objects such as Poachers, Rangers, Gates, etc. There is NO NEED TO write comments.
2. DO NOT write '. . . ' in the program, since it is not a part of the CFG.
3. Write only the action or the sequence of actions such as 'attack(a)' or 'attack(a) attack(b) attack(c)' where a, b and c are positive integers.

Now I have the following strategy program for the poachers that satisfies the CFG, written inside '$< strategy - 1 >< /strategy - 1 >$' tag:
$< strategy - 1 >$
attack(1)
$< /strategy - 1 >$

Now your tasks are the following 5:

1. Understand all the symbols of the given CFG in the context of this given 'Poachers and Rangers' environment.
2. Write an improved strategy for rangers that can defeat strategy-1.

3. You must not use any symbols (for example '$S \rightarrow$', '$\rightarrow$', '|', etc.) outside the given CFG. Write only the action or the sequence of action as mentioned in the strategy writing guideline.

4. Replace '. . . ' with contents meant by this symbol if there are any, then write only the new strategy program inside the '$< rangersStrategy >< /rangersStrategy >$' tag.

5. Check the strategy program and ensure it does not violate the rules of the CFG or the guidelines for writing the strategy.

A.3.3 Feedback Attempt

We have an environment called 'Poachers and Rangers' where 2 teams called poachers and rangers are competing with each other in a national park and its surroundings. The park has 60 gates in total. The goal for each team is to defeat the opponents.

Now I have the following CFG to write programs for poachers in the above environment:
CFG for Poachers:

$$S \rightarrow SA \mid A$$
$$A \rightarrow attack(n)$$
$$n \rightarrow 1 \mid 2 \mid 3 \mid \ldots \mid 59 \mid 60$$

The following is the CFG to write programs for rangers:
CFG for Rangers:

$$S \rightarrow SA \mid A$$
$$A \rightarrow defend(n)$$
$$n \rightarrow 1 \mid 2 \mid 3 \mid \ldots \mid 59 \mid 60$$

The following is the explanation of the above CFG:
CFG Explanation:

S: Starting symbol that can contain one or multiple actions.

A: Refers to the action taken by the team.

attack(n): Refers to the action to attack the n-th gate of the park

defend(n): Refers to the action to defend the n-th gate of the park

n: Any positive integer up to 60.

. . . : It is not part of the CFG. It has been used to indicate all positive numbers in between.

The following are some guidelines for writing the strategy program:
Strategy Writing Guidelines:

1. There is NO NEED TO write classes or initiate objects such as Poachers, Rangers, Gates, etc. There is NO NEED TO write comments.
2. DO NOT write '. . . ' in the program, since it is not a part of the CFG.
3. Write only the action or the sequence of actions such as 'attack(a)' or 'attack(a) attack(b) attack(c)' where a, b and c are positive integers.

Now I have the following strategy program for the poachers that satisfies the CFG, written inside '$< strategy - 1 >< /strategy - 1 >$' tag:
$< strategy - 1 >$
attack(1)
$< /strategy - 1 >$

Here is a strategy written inside ¡strategy2¿¡/strategy2¿ tag for rangers that failed to defeat the given poachers' strategy1:
$< strategy - 2 >$
defend(60)
$< /strategy - 2 >$

The rangers following this strategy-2 defended 1 gate(s), but could not defend gate 1, whereas, the poachers following the strategy-1 attacked gate 1.
Now your tasks are the following 6:

1. Understand all the symbols of the given CFG in the context of this given 'Poachers and Rangers' environment.
2. Analyze why strategy-2 could not defeat strategy-1.
3. Write an improved strategy-2 for rangers that can defeat strategy-1.
4. You must not use any symbols (for example '$S \rightarrow$', '$\rightarrow$', '|', etc.) outside the given CFG. Write only the action or the sequence of action as mentioned in the strategy writing guideline.
5. Replace '. . . ' with contents meant by this symbol if there are any, then write only the new strategy program inside the '$< rangersStrategy >< /rangersStrategy >$' tag.
6. Check the strategy program and ensure it does not violate the rules of the CFG or the guidelines for writing the strategy.

A.4 Climbing Monkey Prompts

A.4.1 Initial Attempt

We have an environment called 'Climbing Monkey' where 2 monkeys are competing with each other to climb a tree. The tree has an infinite number of branches. The goal for one monkey is to defeat another monkey.

Now I have the following CFG to write programs for the above environment:
CFG:

$$S \rightarrow SA \mid A$$
$$A \rightarrow climb(n)$$
$$n \rightarrow 1 \mid 2 \mid 3 \mid \ldots \mid infinity$$

The following is the explanation of the above CFG:
CFG Explanation:

S: Starting symbol that can contain one or multiple actions

A: Refers to the action taken by the monkey.

climb(n): Refers to the action to climb the n-th branch of the tree.

n: Any positive integer upto infinity.

. . . : It is not part of the CFG. It has been used to indicate all positive numbers in between.

The following are some guidelines for writing the strategy program:
Strategy Writing Guidelines:

1. There is NO NEED TO write classes or initiate objects such as Monkey, Tree, etc. There is NO NEED TO write comments.
2. DO NOT write '. . . ' in the program, since it is not a part of the CFG.
3. Write only the action or the sequence of actions such as 'climb(a)' or 'climb(a) climb(b) climb(c)' where a, b and c are positive integers.

Now your tasks are the following 5:

1. Understand all the symbols of the given CFG in the context of this given 'Climbing Monkey' environment.
2. Write a very strong strategy considering the given environment, the above CFG and the strategy writing guidelines.

3. You must not use any symbols (for example '$S \rightarrow$', '$\rightarrow$', '|', '. . . ' etc.) outside the given CFG. Remove these symbols if there are any.

4. Replace '. . . ' with contents meant by this symbol if there are any, then write only the strategy program inside '$< strategy >< /strategy >$' tag.

5. Check the strategy program and ensure it does not violate the rules of the CFG or the guidelines for writing the strategy.

A.4.2 First Attempt

We have an environment called 'Climbing Monkey' where 2 monkeys are competing with each other to climb a tree. The tree has an infinite number of branches. The goal for one monkey is to defeat another monkey.

Now I have the following CFG to write programs for the above environment:
CFG:

$$S \rightarrow SA \mid A$$
$$A \rightarrow climb(n)$$
$$n \rightarrow 1 \mid 2 \mid 3 \mid \ldots \mid infinity$$

The following is the explanation of the above CFG:
CFG Explanation:

S: Starting symbol that can contain one or multiple actions

A: Refers to the action taken by the monkey.

climb(n): Refers to the action to climb the n-th branch of the tree.

n: Any positive integer upto infinity.

. . . : It is not part of the CFG. It has been used to indicate all positive numbers in between.

The following are some guidelines for writing the strategy program:
Strategy Writing Guidelines:

1. There is NO NEED TO write classes or initiate objects such as Monkey, Tree, etc. There is NO NEED TO write comments.

2. DO NOT write '. . . ' in the program, since it is not a part of the CFG.

3. Write only the action or the sequence of actions such as 'climb(a)' or 'climb(a) climb(b) climb(c)' where a, b and c are positive integers.

Now I have the following strategy program that satisfies the CFG, written inside '$< strategy - 1 >< /strategy - 1 >$' tag:
$< strategy - 1 >$
climb(1) climb(2) climb(3)
$< /strategy - 1 >$

Now your tasks are the following 5:

1. Understand all the symbols of the given CFG in the context of this given 'Climbing Monkey' environment.
2. Write an improved strategy that will help the monkey defeat another monkey following strategy-1.
3. You must not use any symbols (for example '$S \rightarrow$', '$\rightarrow$', '|', etc.) outside the given CFG. Write only the action or the sequence of action as mentioned in the strategy writing guideline.
4. Replace '. . . ' with contents meant by this symbol if there are any, then write only the new strategy program inside the '$< newStrategy >< /newStrategy >$' tag.
5. Check the strategy program and ensure it does not violate the rules of the CFG or the guidelines for writing the strategy.

A.4.3 Feedback Attempt

We have an environment called 'Climbing Monkey' where 2 monkeys are competing with each other to climb a tree. The tree has an infinite number of branches. The goal for one monkey is to defeat another monkey.

Now I have the following CFG to write programs for the above environment:
CFG:

$$S \rightarrow SA \mid A$$
$$A \rightarrow climb(n)$$
$$n \rightarrow 1 \mid 2 \mid 3 \mid \ldots \mid infinity$$

The following is the explanation of the above CFG:
CFG Explanation:

S: Starting symbol that can contain one or multiple actions

A: Refers to the action taken by the monkey.

climb(n): Refers to the action to climb the n-th branch of the tree.

n: Any positive integer upto infinity.

. . . : It is not part of the CFG. It has been used to indicate all positive numbers in between.

The following are some guidelines for writing the strategy program:
Strategy Writing Guidelines:

1. There is NO NEED TO write classes or initiate objects such as Monkey, Tree, etc. There is NO NEED TO write comments.
2. DO NOT write '. . . ' in the program, since it is not a part of the CFG.
3. Write only the action or the sequence of actions such as 'climb(a)' or 'climb(a) climb(b) climb(c)' where a, b and c are positive integers.

Now I have the following strategy program that satisfies the CFG, written inside '$< strategy - 1 >< /strategy - 1 >$' tag:
$< strategy - 1 >$
climb(1) climb(2) climb(3)
$< /strategy - 1 >$

Here is a strategy that could not improve the given strategy written inside '$< strategy - 2 >< /strategy - 2 >$' tag:
$< strategy - 2 >$
climb(1) climb(2) climb(4) climb(8) climb(16)
$< /strategy - 2 >$
The monkey following strategy2 could climb 2 branches, whereas, the opponent monkey following strategy1 could climb 3 branches.

Now your tasks are the following 6:

1. Understand all the symbols of the given CFG in the context of this given 'Climbing Monkey' environment.
2. Analyze why strategy-2 above could not improve strategy-1.
3. Write an improved strategy-2 that will help the monkey defeat another monkey following strategy-1.
4. You must not use any symbols (for example '$S \rightarrow$', '$\rightarrow$', '|', etc.) outside the given CFG. Write only the action or the sequence of action as mentioned in the strategy writing guideline.

5. Replace '. . . ' with contents meant by this symbol if there are any, then write only the new strategy program inside the '$<$ *newStrategy* $><$ */newStrategy* $>$' tag.

6. Check the strategy program and ensure it does not violate the rules of the CFG or the guidelines for writing the strategy.