LEARNING BEHAVIOURAL ABSTRACTIONS FOR PROGRAMMATIC POLICIES

Anonymous authors
Paper under double-blind review

ABSTRACT

Synthesizing a programmatic policy is a search problem over a discrete, non-smooth space. Small changes to a program can produce large changes in its behaviour and return. In practice, the resulting optimization landscape tends to be flat, making it difficult to know where to search next. Existing approaches provide search guidance through domain-specific, handcrafted behavioural features, which limit which behaviours the search can explicitly target to those their designer chose to describe. We introduce VIES (Video-Induced Exploration Space), a method that turns execution trajectories in the form of videos into a discrete behaviour abstraction space. From unlabelled videos of agents acting in the environment, a vector-quantized model maps executions to short sequences of discrete symbols. These symbols form a finite, factorized behavioural abstraction that can guide search when the task outcome cannot distinguish candidates, replacing behavioural features that otherwise have to be designed by hand for each domain. The representation is independent of how candidate programs are constructed: we use the same learned behavioural signal to guide synthesis over both a domain-specific language and LLM-written Python. We evaluate the approach on MicroRTS and Can't Stop, across best-response and self-play settings. Comparing conditions with and without the discrete abstractions shows that access to the trajectory alone is not enough: guidance from raw videos captures little of the benefit provided by the discrete abstraction. Surprisingly, even an untrained quantizer can be helpful, while training organizes the resulting space around more meaningful visual, behavioural, and outcome-related structure. Together, these results show that execution traces in the form of video can provide the behavioural structure needed to navigate otherwise flat program spaces without requiring that structure to be specified by hand.

1 INTRODUCTION

Programmatic representations allow a learned policy to be examined as a set of instructions for choosing actions in sequential decision-making problems. With a suitable language, a person can understand and modify a program encoding a policy. Moreover, such a representation allows for verification (Verma et al., 2018; Bastani et al., 2018). Learning policies in this form has been studied in control problems (Verma et al., 2019; Qiu & Zhu, 2022) and strategy games (Mariño et al., 2021; Medeiros et al., 2022). Language models have also made it possible to generate policy programs from descriptions of the desired behaviour (Liang et al., 2023; Liu et al., 2025).

The main issue is that finding such a program poses a difficult optimization problem. The program space is discrete and non-differentiable, and changes to a program can cause abrupt changes in behaviour. To make things worse, most programs are evaluated equally, creating flat regions of the optimization landscape. In this paper we leverage that policy evaluation provides information that can help guide the synthesis process. In addition to a return specifying how well a program performed at a given task, an evaluation produces an execution that records how the policy interacts with the environment. We investigate how to learn a representation of executions that can provide useful guidance to the process of searching for programs to solve sequential decision-making problems.

Our starting point is Bilevel Synthesis (Bi-S), which uses behavioural targets to direct program search (Aleixo & Lelis, 2023). Bi-S represents behaviour with a vector of game features and learns which combinations of feature values are associated with higher returns. It then uses these combinations as

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

targets for a search that seeks programs with behaviour similar to the target. This approach leaves an important part of the problem to the designer: deciding which behaviours the search should target.

We investigate whether the behavioural space required by this search can be learned from videos of policy executions. Videos can record changes across both space and time without requiring a separate feature for each behaviour of interest. Learning behavioural representations from observations has been explored in quality-diversity methods, including methods that construct discrete spaces through vector quantization (Grillotti & Cully, 2022; Tsakonas & Chatzilygeroudis, 2025). Our question concerns the use of such a representation in program synthesis: can it provide targets that help search find better programs, without a designer first specifying the behaviours those targets should describe?

We propose Video-Induced Exploration Space (VIES). Given unlabelled execution videos, VIES learns a vector-quantized representation using a video encoder and a finite codebook (van den Oord et al., 2017; Yu et al., 2023). An execution is represented by a sequence of code indices. The possible values at each sequence position define the choices for one behavioural variable, so combinations of these values form a discrete, factorized search space. This gives Bi-S a learned replacement for its handcrafted feature space. The representation can also compare a program's execution with a reference execution, providing guidance without an outer search over behavioural targets.

VIES operates on the videos produced by programs, which allows the behavioural space to be used with different program representations. We use it with both grammar-based synthesis and Python programs written by a large language model. In best-response synthesis, the target comes from a reference policy playing against a fixed opponent. In self-play, targets are selected through the outer search of Bi-S. In both uses, the synthesis procedure constructs the programs and evaluates their returns; VIES supplies the additional comparison between observed and target behaviour. In our evaluation, the return of the policy remains the primary search criterion, with behavioural similarity guiding choices when the returns do not distinguish programs, i.e., it breaks ties.

Our experiments in MicroRTS and Can't Stop evaluate both the effect on synthesis and the information represented by VIES. Guidance from discrete video representations improves search with programs written in a domain-specific language or in a general-purpose language. In self-play, VIES achieves higher mean win rates than the handcrafted and unguided baselines. Surprisingly, our experiments show that even an untrained model provides useful guidance. Training improves search in MicroRTS, although the trained and untrained conditions perform similarly in Can't Stop. Analysis of the MicroRTS representations shows that learning organizes the space around visual and behavioural properties, including attack timing and unit spread, and makes game outcomes more predictable from the representation. These results suggest that execution videos can offer the behavioural structure needed by program search without specifying that structure through handcrafted features.

2 Problem Formulation

We consider sequential decision-making problems modelled as Markov decision processes (MDPs). An MDP is a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, r, \mu)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $P(s' \mid s, a)$ is the transition probability from state s to state s' after action a, $r(s, a, s')$ is the associated reward, and μ is a distribution over initial states. A policy π maps each state to a distribution over actions.

Executing a policy π in $\mathcal{M}$ induces a probability distribution $\mathbb{P}_{\pi,\mathcal{M}}$ over trajectories. A trajectory is $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T)$, where $s_0 \sim \mu$. Its probability under π and $\mathcal{M}$ is $\mathbb{P}_{\pi,\mathcal{M}}(\tau) = \mu(s_0) \prod_{t=0}^{T-1} \pi(a_t \mid s_t) P(s_{t+1} \mid s_t, a_t)$. We define the return of a trajectory as $U(\tau) = \sum_{t=0}^{T-1} r(s_t, a_t, s_{t+1})$. The expected return of a policy is $J(\pi) = \mathbb{E}_{\tau \sim \mathbb{P}_{\pi,\mathcal{M}}}[U(\tau)]$.

We focus on programmatic policies. Let $\mathcal{P}$ denote a space of executable programs, where each program $p \in \mathcal{P}$ encodes a policy π_p. Our formulation does not depend on how $\mathcal{P}$ is represented or how candidates are generated. For example, programs may be drawn from a domain-specific language or generated by a language model. Program synthesis seeks $p^* = \arg\max_{p \in \mathcal{P}} J(\pi_p)$.

The formulation also covers interactions with another agent. Suppose an opponent follows a fixed policy π^-, and let $P_G(s' \mid s, a, a^-)$ denote the transition model of the underlying interaction, where a^- is the opponent's action. From the perspective of the policy being synthesized, the opponent can be absorbed into the environment dynamics $P_{\pi^-}(s' \mid s, a) = \sum_{a^-} \pi^-(a^- \mid s) P_G(s' \mid s, a, a^-)$.

A synthesis problem against a fixed opponent can be treated as an MDP as defined above. If the opponent policy changes over time, each fixed-opponent interaction induces a corresponding MDP.

3 BACKGROUND

This section introduces the components needed to understand VIES. We first describe local search for program synthesis (Section 3.1), then review Bi-S (Section 3.2) and the VQ-VAE model used to learn the behavioural abstractions from video (Section 3.3).

3.1 LOCAL SEARCH FOR PROGRAM SYNTHESIS

We consider local search algorithms for synthesizing programs. In local search, we start with a candidate program and, in every iteration, we generate a set of neighbours by changing small parts of the current program (Mariño et al., 2021; Aleixo & Lelis, 2023). Each neighbour is executed in the environment, and its evaluation determines whether it becomes the current candidate. In hill climbing, a neighbour replaces the current program only if it has a higher estimated return; otherwise, the search keeps the current program and tries another change. This process repeats until the search budget is exhausted. Changes can be generated according to the rules of a domain-specific language or proposed by a language model. Bi-S uses local search with a behaviour-based evaluation function.

3.2 BILEVEL SYNTHESIS

Aleixo & Lelis (2023) introduce Bi-S, which searches concurrently in a behavioural space and in the program space. Let $X_1, \ldots, X_n$ be behavioural variables with finite domains $D_1, \ldots, D_n$, defining the behavioural space $\mathcal{V} = D_1 \times \cdots \times D_n$. A vector $v = (v_1, \ldots, v_n) \in \mathcal{V}$ is a complete assignment to these variables, where $v_i \in D_i$ is the value assigned to X_i. At each outer iteration, the behavioural search proposes a target $v^\star \in \mathcal{V}$, and this target is passed to an inner local search over programs. The inner search searches for a program with high return. When two programs obtain the same return, the search prefers the one whose observed behavioural vector is closer to $v^\star$. The outer search therefore determines which behavioural region to pursue, while the inner search remains entirely in the program space and attempts to realize a high-performing program in that region.

Bi-S maintains a library of programs encountered during search together with their observed behavioural vectors. When the outer search proposes $v^\star$, the program in the library whose observed behaviour is most similar to the target is used to initialize the inner local search. This allows the behavioural search to move the synthesis process between different regions of the program space without modifying the program representation or how the search is performed. The inner search evaluates neighbouring programs, and the programs encountered during this search are added to the library. In this way, the library lets the outer behavioural search effectively redirect, or "jump," the inner search toward regions whose previously observed behaviour is closer to the current target.

The outer search is formulated as a combinatorial multi-armed bandit (CMAB) problem (Gai et al., 2010). Each behavioural coordinate is treated as a variable whose possible values form its arms, and a complete assignment of values forms a macro-arm, corresponding to a behavioural target. Aleixo & Lelis (2023) use Naive Sampling (Ontañón, 2017) to search this combinatorial space. The outer search biases the inner program search by offering a target $v^\star$, and the inner search biases the outer search by informing which values of $v^\star$ are correlated with a higher return.

Appendix A.1 provides pseudocode for the bilevel procedure and its inner program search, and Appendix A.2 describes the Naive Sampling mechanism used to generate behavioural targets.

3.3 VQ-VAE AND DISCRETE VIDEO REPRESENTATIONS

VIES uses the discrete video representation described in this section to induce, from unlabelled execution videos, the behavioural space required by the bilevel search described in Section 3.2.

Vector-quantized autoencoders transform high-dimensional inputs into discrete latent representations. In a VQ-VAE (van den Oord et al., 2017), an encoder maps an input x to a continuous latent representation, which is discretized using a learned codebook $\mathcal{Z} = \{e_1, \ldots, e_K\}$, where each $e_k \in \mathbb{R}^d$ is an embedding vector. For an encoder output $z \in \mathbb{R}^d$, vector quantization selects the

closest codebook entry $k = \arg\min_{j \in \{1,\dots,K\}} \|z - e_j\|_2$, and replaces z with e_k. The index $k \in \{1, \dots, K\}$ therefore represents the continuous latent using one symbol from a finite vocabulary. A decoder network reconstructs the input from the resulting quantized representation.

MAGVIT (Yu et al., 2023) extends vector quantization to video using a three-dimensional tokenizer that preserves spatiotemporal structure. Given a video clip $x \in \mathbb{R}^{T \times H \times W \times 3}$, a 3D encoder maps the video to a lower-resolution spatiotemporal latent grid of dimensions $\tilde{T} \times \tilde{H} \times \tilde{W}$. Because the encoder operates with three-dimensional convolutions, each latent position summarizes space and time information. Each latent vector is quantized using a finite codebook, producing a grid of discrete visual tokens; a corresponding 3D decoder reconstructs the video from the quantized representation.

Let $z_{t,h,w} \in \mathbb{R}^d$ denote the latent vector at temporal position t, height position h, and width position w of this grid, with $t \in \{1, \dots, \tilde{T}\}$, $h \in \{1, \dots, \tilde{H}\}$, and $w \in \{1, \dots, \tilde{W}\}$. Quantization assigns each $z_{t,h,w}$ a code index from $\{1, \dots, K\}$. Flattening the quantized grid yields a sequence of $R = \tilde{T}\tilde{H}\tilde{W}$ discrete code indices, each taking a value in $\{1, \dots, K\}$. Each position in this sequence corresponds to a location in the spatiotemporal latent grid. This preserves the organization induced by the 3D encoder while converting the original high-dimensional video into a compact discrete representation.

4 VIES — Video-Induced Exploration Space

In this section we describe VIES, our system that uses the same Bi-S search, but uses a VQ-VAE to learn the behavioural variables $X_1, \dots, X_n$ and their finite domains $D_1, \dots, D_n$ from videos.

4.1 Overview

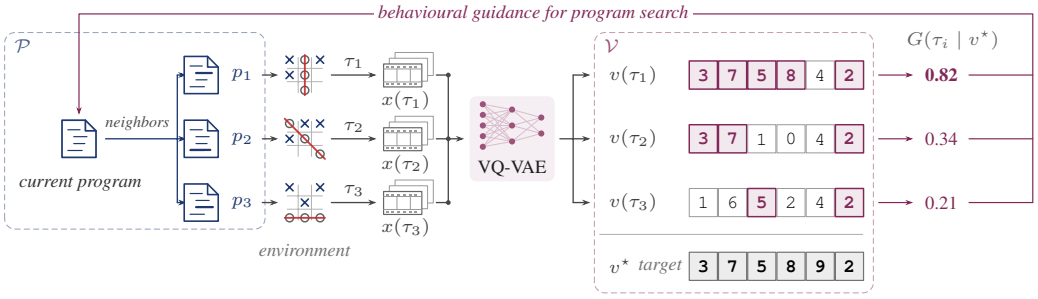


Figure 1: Overview of VIES. Candidate neighbor programs $p_i \in \mathcal{P}$ are executed in the environment, producing trajectories τ_i that are rendered as videos. A vector-quantized video tokenizer maps each video to a discrete behavioural signature $v(\tau_i) \in \mathcal{V}$. The resulting signatures are compared with a behavioural target $v^\star$, providing a behavioural guidance signal for program search. The figure depicts program-space search for a given behavioural target $v^\star$. The target-generation mechanism is not shown: $v^\star$ may be obtained from an oracle/reference execution or proposed by a search algorithm over the behavioural space. Candidate generation remains in the original program space $\mathcal{P}$.

VIES constructs the space of behaviours from unlabelled execution videos using the representation of Section 3.3. A vector-quantized tokenizer maps the video produced by a trajectory of a program to a sequence of discrete tokens, which serves as the signature of the program. The tokenizer is trained from execution video only, without task rewards, action labels, or handcrafted behavioural annotations. The search then uses the resulting signatures as part of the evaluation function. In Figure 1 all three neighbours p_1, p_2, and p_3 have the return of zero (they all represent loses in a game). The search then prefers neighbour p_1 because its signature is closer to the target signature $v^\star$.

4.2 Behaviour Abstraction Space

Let $p \in \mathcal{P}$ be a program and $\tau \sim \mathbb{P}_{\pi_p, \mathcal{M}}$ an observed execution trajectory of its policy π_p. Let $x(\tau)$ denote the video rendered from τ. VIES maps this video to a discrete behavioural signature through a vector-quantized video tokenizer Φ, yielding $v(\tau) = \Phi(x(\tau)) = (v_1, \dots, v_R)$, where

$v_r \in \{1, \ldots, K\}$, R is the number of positions in the tokenizer's latent grid and K is the size of its codebook. The resulting behaviour abstraction space is therefore $\mathcal{V} = \{1, \ldots, K\}^R$. Each observed execution is associated with a point $v(\tau)$ in $\mathcal{V}$. Because a program may induce different trajectories across executions, it may also produce different signatures.

Each coordinate v_r in $v(\tau)$ corresponds to a position in the tokenizer's spatiotemporal latent grid and can take one of the K codebook values. The coordinates therefore inherit the spatiotemporal organization of the tokenizer's latent grid, but they are not assigned manually defined behavioural meanings. Together, the R coordinates give $\mathcal{V}$ a finite structure in which different combinations of code values represent different regions of observed behaviour. The structure of the behaviour abstraction space is thus induced from execution data instead of specified through handcrafted behavioural features, as in previous work (Aleixo & Lelis, 2023).

The codebook also provides a notion of similarity within $\mathcal{V}$. Let e_c denote the embedding associated with code c, and define the similarity between two code values as $M_{c,c'} = \frac{1}{2}(1 + \cos(e_c, e_{c'}))$. This rescaling gives $M_{c,c'} = 1$ for identical embeddings, $M_{c,c'} = 0.5$ for orthogonal embeddings, and $M_{c,c'} = 0$ for embeddings pointing in opposite directions. For two signatures $v, v' \in \mathcal{V}$, we compare corresponding positions and average their code similarities: $\mathrm{sim}(v, v') = \frac{1}{R}\sum_{r=1}^{R} M_{v_r, v'_r}$. Identical codes contribute the maximum similarity, while different codes can still be considered close when their embeddings are similar. VIES can therefore preserve the discrete distinctions introduced by quantization while also providing a similarity metric between different signatures.

4.3 Behavioural Targets

VIES uses the behaviour abstraction space through a target signature $v^\star \in \mathcal{V}$. A target may be the signature of a previously observed execution. For example, in best-response synthesis, a reference policy can be executed against the opponent, and the resulting execution video can be mapped to a target signature $v^\star$. In this case, the agent would learn to play while trying to explore behaviours similar to those of its opponent. Alternatively, $v^\star$ may be proposed directly in $\mathcal{V}$ by the CMAB-based outer search used in the bilevel procedure described in Section 3.2. In the latter case, the structure of $\mathcal{V}$ allows a search procedure to propose targets by choosing values for individual token positions. This requires a finite, factorized space whose coordinates serve as the variables of the CMAB and whose values serve as its arms; the signature provides this structure with a small number of variables (R), each with a small domain (K). Raw pixels would instead yield hundreds of thousands of variables that individually carry little behavioural information, making independently chosen per-pixel targets meaningless. A constructed target need not correspond to a previously observed execution, and it need not be directly realizable by a program: not every point in $\mathcal{V}$ corresponds to a behaviour that some program can produce. It specifies a direction for behavioural exploration in the learned space. After a candidate program is evaluated and a trajectory τ is observed, VIES measures how closely that execution approaches the current target as $G(\tau \mid v^\star) = \mathrm{sim}(v(\tau), v^\star)$.

The return J and the similarity G play different roles: J measures program quality, while G provides behavioural search guidance toward $v^\star$. This guidance does not redefine return; it provides a behavioural direction that a search procedure can use while exploring the program space. Section 4.4 describes how our search procedures combine G with J.

4.4 Using VIES in Search

We use VIES in two settings: best-response synthesis against a fixed opponent, which corresponds to an MDP (Section 2), and self-play, in which the opponent changes over time, so it corresponds to a sequence of MDPs. In both settings, behavioural guidance uses only the trajectory of the synthesized program, so signatures are always computed from the candidate's own perspective.

Combining return and similarity. The inner search is the local search of Section 3.1, with G as a secondary criterion. While Bi-S uses behaviour only when two programs obtain the same return, we use it whenever their returns fall within a comparison band, which accommodates noisy return estimates. Let p be the current program, with trajectory τ, and let p' be a neighbour, with trajectory τ'. If $J(p)$ and $J(p')$ differ by more than a comparison band, p' is accepted if and only if $J(p') > J(p)$. Otherwise, p' is accepted with probability $\min\{1, \exp(\beta\, \Delta G/T)\}$, where $\Delta G = G(\tau' \mid v^\star) - G(\tau \mid v^\star)$, $\beta > 0$ is a constant, and T is a temperature that decreases

over iterations. Return therefore remains the primary criterion: similarity never overrides a return difference outside the band. Algorithm 2 in Appendix A.1 gives the complete procedure.

Game-based synthesis. When computing a best response to a target policy, we use $v^\star$ as the signature of a reference policy executed against the opponent, and the inner search is run directly with this target, without the outer CMAB. When learning how to play a game through self-play, we follow the Fictitious Play algorithm (Brown, 1951), where each player maintains a growing policy pool of all previous versions of the learning agent. Then, in every iteration, we approximate a best response to the current pool of policies; the approximated best response is then added to the pool. Each best response is synthesized with Bi-S (Section 3.2). Behavioural guidance is computed from executions against the most recently added policy in the pool in the fictitious play algorithm.

Stochastic environments. In stochastic environments, a single signature is a noisy description of the program's behaviour. In this case, VIES aggregates the signatures of multiple executions of the program via the median into a single representation; the target is aggregated from several vectors sampled from the outer search, and similarity is computed between the aggregated representations. The comparison band is then set from the uncertainty of the return estimates, so that similarity decides only between candidates whose returns are statistically indistinguishable (Appendix A.4).

5 Experiments

We evaluate VIES along three questions. First, does the VIES behavioural abstraction improve program synthesis? Second, what does the discrete behavioural representation provide beyond handcrafted features and direct raw-video comparison? Third, what does training contribute beyond the behavioural resolution already provided by vector quantization?

5.1 Experimental Setup

MicroRTS. MicroRTS (Ontañón et al., 2018) is a two-player real-time strategy game. We evaluate VIES on three maps: `workers_24x24` and `basesWorkers32x32A`, which support both economic and combat behaviour, and `CombatMap` (16×8), which contains only combat units (four ranged and four heavy units per player) with no resources or structures, making behaviour depend primarily on movement, positioning, and target selection. Best-response synthesis is evaluated against MayariBot, the winner of the Classic track of the 2024 microRTS AI Competition (microRTS AI Competition, 2024). The target $v^\star$ is the signature of the player-0 MayariBot playing with itself.

Can't Stop. Can't Stop is a stochastic board game in which the player chooses which available column to advance and whether to continue rolling or stop. Because dice rolls determine the available actions, executions of the same program can produce different trajectories and outcomes. We evaluate VIES in Can't Stop using a domain-specific language from Medeiros et al. (2022) under self-play.

DSL-based synthesis. In MicroRTS, candidate programs are represented using the grammar-based language from prior work (Moraes et al., 2018; Mariño et al., 2021). Neighbouring programs are generated by mutating subtrees of the current program's abstract syntax tree, preserving syntactic validity. For Can't Stop, we use the language introduced by Medeiros et al. (2022).

LLM-based synthesis. In MicroRTS, we also synthesize Python programs using a language model as the mutation operator within the same local-search framework: at each iteration, the model receives the current program and evaluation feedback and returns one modified candidate (Appendix A.7). For gpt-oss-120b, search budgets are measured in generated programs; for gemini-2.5-flash-lite, the budget is monetary, and $1 corresponds to approximately 800 generated programs on average.

Behaviour Abstraction. We train a MAGVIT-style tokenizer separately for each environment configuration, on videos of existing agents generated independently of the programs synthesized in our experiments. In MicroRTS, we remove videos involving MayariBot before training, so the opponent is unseen during representation training. Training details are in Appendices A.6 and A.5.

Guidance Conditions. We consider five evaluation functions: *No guide* uses return alone; *Handcrafted* uses a domain-specific behavioural representation from prior work; *Raw video* compares execution recordings directly through mean pixelwise L_1 distance; *Untrained VIES* uses a randomly

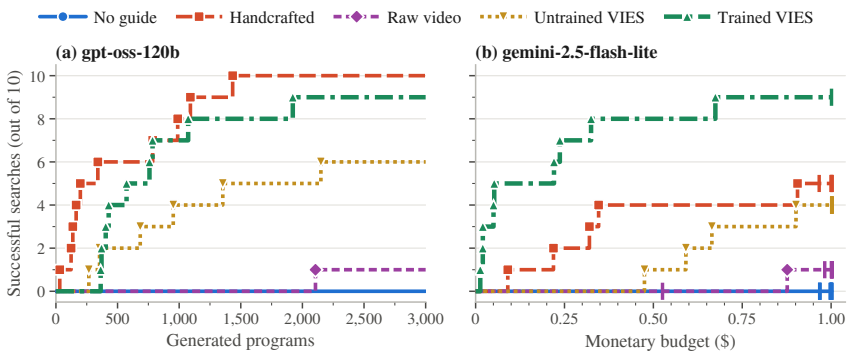


Figure 2: Best-response synthesis in MicroRTS with LLM-based candidate generation. Curves show how many of ten independent searches have found a program that defeats MayariBot as budget increases. (a) gpt-oss-120b, measured in generated programs up to 3,000. (b) gemini-2.5-flash-lite, measured in monetary budget capped at $1; $1 corresponds to approximately 800 generated programs on average in our experiments. Vertical ticks mark searches that stopped without a winning program.

initialized tokenizer with the same architecture and codebook structure as the trained model, kept fixed without representation training; *Trained VIES* uses the tokenizer after training on execution videos. The handcrafted representation consists of seven production counts in MicroRTS (Medeiros et al., 2022; Aleixo & Lelis, 2023) and of the final progress in each board column in Can't Stop (Medeiros et al., 2022); Appendix A.3 gives both, together with details of the Raw video comparison. The Raw video condition is evaluated only in best-response synthesis: treating each pixel as a CMAB variable would give the outer search of Bi-S hundreds of thousands of variables of varying number across recordings (Section 4.3); self-play therefore compares the remaining four conditions.

Evaluation Protocol. In best-response synthesis, we report the cumulative number of independent searches that defeat MayariBot as the budget increases; for DSL-based synthesis, the budget is measured in evaluated programs. In MicroRTS self-play, each condition contributes 30 independently synthesized programs, and at each checkpoint each condition is evaluated against the final strategies of the other three conditions; we report mean win rates and 95% confidence intervals across synthesized programs. In Can't Stop self-play, each condition is evaluated against a fixed pool of the final strategies from the other three conditions; means and 95% confidence intervals are computed across 16 independent runs per condition. Appendix A.4 details the evaluation protocol for Can't Stop.

5.2 Does VIES Improve Program Synthesis?

5.2.1 Best-Response Synthesis

LLM-based synthesis. Figure 2 reports how many of ten independent searches defeat MayariBot (y-axis) per generated programs (x-axis). With both language models, the VIES conditions succeed more often than Raw video (1/10) and No guide (0/10), and Trained VIES succeeds in 9/10 runs. Handcrafted is the strongest condition with gpt-oss-120b (10/10, with a median of 269 generated programs to a win; 666 for Trained VIES), but succeeds in only 5/10 runs with gemini-2.5-flash-lite.

DSL-based synthesis. The same pattern holds with grammar-based synthesis (Figure 3). On `workers_24x24` and `basesWorkers32x32A`, Trained VIES succeeds in all runs, Handcrafted in 9/10 and 8/10, and Untrained VIES in 7/10 and 8/10, while Raw video and No guide never succeed. `CombatMap` is saturated because MayariBot is weak on this map: all conditions succeed in all runs.

5.2.2 Self-Play Synthesis

MicroRTS. Trained VIES achieves the highest mean win rate on all three maps (Figure 4), with 95% confidence intervals disjoint from those of every other condition on `basesWorkers32x32A` and `CombatMap`. Surprisingly, Untrained VIES is the second-best condition on all three maps, ahead of Handcrafted. On `CombatMap`, Handcrafted even falls below No guide, since none of the seven production features capture the movement behaviour needed on this map.

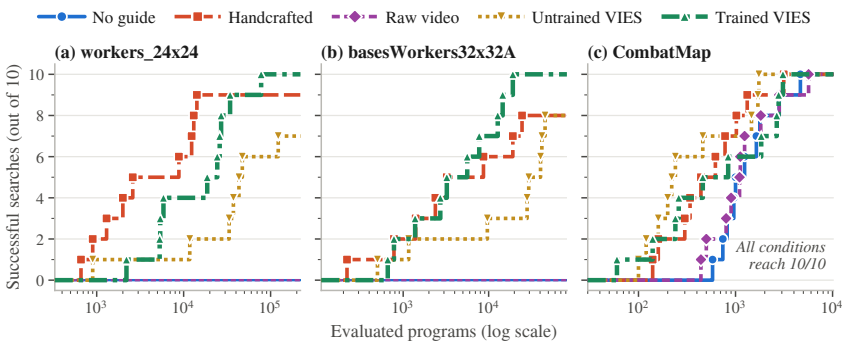


Figure 3: Best-response synthesis in MicroRTS with grammar-based (DSL) candidate generation: number of ten independent searches that have found a program defeating MayariBot.

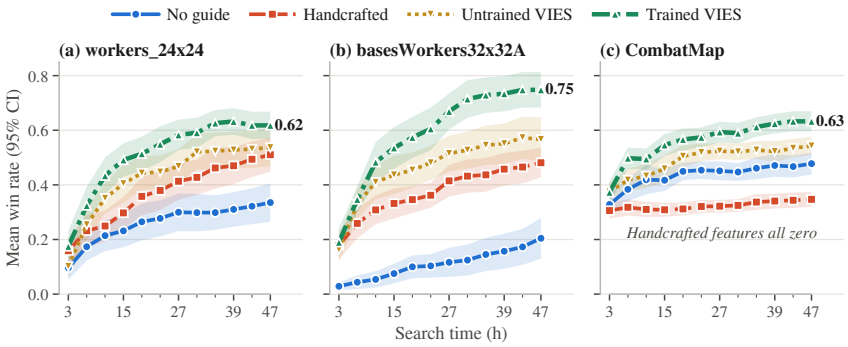


Figure 4: Self-play synthesis in MicroRTS. Curves show mean win rate against the final 47-hour strategies of the other three conditions, with 95% confidence intervals across 30 independently synthesized programs per condition.

Can't Stop. Trained and Untrained VIES perform nearly identically, and both outperform Handcrafted and No guide in win rate and round-robin finishing position (Figure 5).

Why does Untrained VIES perform so well? Even with random weights, the encoder maps each spatiotemporal region of a video to one of K codes, so executions that differ in where and when events occur tend to receive different signatures. To measure this, we encode the 3,884 distinct videos produced by gpt-oss-120b best-response synthesis under No guide on `workers_24x24`: the seven handcrafted features yield only 974 distinct signatures, while trained and untrained tokenizers yield 3,696–3,728 and 3,802–3,818 distinct signatures, respectively. The model does not need to assign meaning to its codes for them to be useful: search can learn which code values are associated with higher return, as the CMAB of Bi-S does. Since untrained models yield even more distinct signatures than trained ones, training's advantage cannot be explained by higher resolution.

What does training add? To isolate the effect of training, we compare four trained and six untrained models on a `workers_24x24` self-play corpus. First, training organizes the codes around visual structure: signatures predict the time-averaged frames of an execution with $R^2 = 0.17$–0.21 for trained models, compared with 0.05–0.06 for untrained ones. Second, trained codes align with behaviours that the handcrafted features do not describe: all four trained tokenizers select a latent position most associated with attack timing and another as most associated with spatial spread. Third, trained signatures are more predictive of game outcome, with AUCs of 0.72–0.80 compared with 0.56–0.69 for untrained ones. While an untrained model separates executions, training arranges this separation around structure that is behavioural and related to outcome.

6 More Related Work

Latent program spaces. LEAPS searches a program embedding space learned by a variational autoencoder (Trivedi et al., 2021), HPRL learns a meta-policy that composes programs from such a

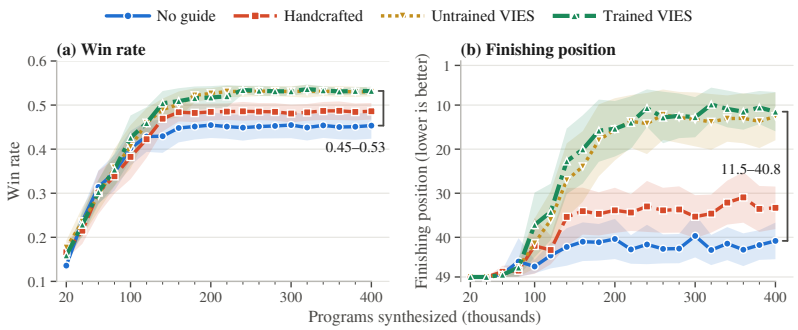


Figure 5: Self-play synthesis in Can't Stop. (a) Win rate over 100 games against each of the 48 final policies of the other three conditions. (b) Finishing position in a closed 49-player round robin against the final policies. Means and 95% confidence intervals for 16 independent runs per condition.

space (Liu et al., 2023), and Latent Programmer predicts discrete latent codes before generating a program (Hong et al., 2021). These methods learn representations of programs, whereas VIES learns a representation of executions. Carvalho et al. (2024) show that direct program search can outperform LEAPS and HPRL, which motivates keeping search in the original program space.

Learned behavioural spaces. Quality-diversity methods such as novelty search and MAP-Elites rely on behavioural descriptors to measure diversity or organize an archive (Lehman & Stanley, 2011; Mouret & Clune, 2015; Cully et al., 2015). Several approaches learn these descriptors instead of specifying them: AURORA from visited states (Cully, 2019; Grillotti & Cully, 2022), VQ-Elites through vector quantization (Tsakonas & Chatzilygeroudis, 2025), and AutoQD from policy occupancy measures (Hedayatian & Nikolaidis, 2026); Go-Explore similarly maps states to discrete cells for exploration (Ecoffet et al., 2021). VIES also learns behavioural structure, but uses the learned space to set targets for search instead of populating a repertoire.

Reward shaping and learning from video. Reward shaping adds guidance when task rewards are sparse (Ng et al., 1999). Learned video representations support world models (Ha & Schmidhuber, 2018; Micheli et al., 2023), latent-action discovery (Bruce et al., 2024), and video-derived learning signals for reinforcement learning and robotics (Sermanet et al., 2018; Zakka et al., 2021; Nair et al., 2022; Fan et al., 2022; Escontrela et al., 2023). Closest to our work, Aytar et al. (2018) learn an embedding from unlabelled game footage and use distances in it to guide an agent. These methods target neural policies and turn the representation into a reward; VIES compares complete executions through discrete signatures and uses their similarity to guide program search while return remains the objective. Importantly, VIES learns the relationship of the learned abstractions with return.

7 Conclusion

Synthesizing programmatic policies is difficult in part because the search landscape is flat: many neighbouring programs obtain the same return, and return alone does not indicate where to search next. Bi-S addresses this with behavioural targets, but it requires a designer to decide which behaviours those targets describe. We asked whether this behavioural space could instead be learned from videos of program executions. VIES shows that it can: a vector-quantized video tokenizer turns each execution into a sequence of discrete codes, and the resulting finite, factorized space serves both as a learned replacement for the handcrafted features of Bi-S. Our experiments show that, in self-play, Trained VIES achieved the highest win rates on all three MicroRTS maps and, together with Untrained VIES, outperformed the handcrafted and unguided baselines in Can't Stop. Even an untrained tokenizer separates executions more finely than handcrafted features, which is enough for search to learn which codes are associated with higher return. Training organizes into behavioural structures that the handcrafted features do not capture and that is more predictive of game outcome.

Limitations. The results also point to limitations. Handcrafted features remained competitive in some settings, most notably LLM-based synthesis with gpt-oss-120b, and training did not improve over the untrained tokenizer in Can't Stop, where the tokenizer corpus contained only three fixed players. Understanding how this corpus diversity shapes the learned space is a natural next step.

AI USE STATEMENT

We used generative AI tools to assist with software development and the preparation of this manuscript. In particular, LLMs were used to help write, debug, and revise experimental code, and to assist with drafting, editing, and improving the clarity and presentation of the paper. All AI-assisted code was reviewed by the authors and validated through testing and experimental execution. AI-assisted text was reviewed and revised by the authors, and scientific claims, results, and citations were checked against the underlying experiments and relevant sources. The authors made the final decisions regarding the methodology, experimental design, analysis, and presentation of the work and take responsibility for all content in this paper.

ETHICS STATEMENT

This work studies methods for synthesizing programmatic policies in simulated reinforcement-learning environments. Our experiments do not involve human subjects or personal data. While automated program synthesis and decision-making systems may have broader societal implications, we do not evaluate or deploy the proposed method in safety-critical or real-world decision-making settings.

REFERENCES

David S. Aleixo and Levi H. S. Lelis. Show me the way! Bilevel search for synthesizing programmatic strategies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2023.

Yusuf Aytar, Tobias Pfaff, David Budden, Thomas Paine, Ziyu Wang, and Nando de Freitas. Playing hard exploration games by watching YouTube. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

GW Brown. Iterative solution of games by fictitious play. *Activity Analysis of Production and Allocation (TC Koopmans, Ed.)*, pp. 374–376, 1951.

Jake Bruce, Michael D. Dennis, Ashley Edwards, Jack Parker-Holder, Yuge Shi, Edward Hughes, Matthew Lai, Aditi Mavalankar, Richie Steigerwald, Chris Apps, Yusuf Aytar, Sarah Bechtle, Feryal Behbahani, Stephanie Chan, Nicolas Heess, Lucy Gonzalez, Simon Osindero, Sherjil Ozair, Scott Reed, Jingwei Zhang, Konrad Zolna, Jeff Clune, Nando de Freitas, Satinder Singh, and Tim Rocktäschel. Genie: Generative interactive environments. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2024.

Tales H. Carvalho, Kenneth Tjhia, and Levi H. S. Lelis. Reclaiming the source of programmatic policies: Programmatic versus latent spaces. In *International Conference on Learning Representations (ICLR)*, 2024.

Antoine Cully. Autonomous skill discovery with quality-diversity and unsupervised descriptors. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2019.

Antoine Cully, Jeff Clune, Danesh Tarapore, and Jean-Baptiste Mouret. Robots that can adapt like animals. *Nature*, 521(7553):503–507, 2015.

Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. First return, then explore. *Nature*, 590(7847):580–586, 2021.

Alejandro Escontrela, Ademi Adeniji, Wilson Yan, Ajay Jain, Xue Bin Peng, Ken Goldberg, Youngwoon Lee, Danijar Hafner, and Pieter Abbeel. Video prediction models as rewards for reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. MineDojo: Building open-ended embodied agents with internet-scale knowledge. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2022.

Yi Gai, Bhaskar Krishnamachari, and Rahul Jain. Learning multiuser channel allocations in cognitive radio networks: A combinatorial multi-armed bandit formulation. In *2010 IEEE Symposium on New Frontiers in Dynamic Spectrum (DySPAN)*, pp. 1–9. IEEE, 2010.

James Glenn and Christian Aloi. A generalized heuristic for Can't Stop. In *Proceedings of the 22nd International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, pp. 421–426, 2009.

Luca Grillotti and Antoine Cully. Unsupervised behaviour discovery with quality-diversity optimisation. *IEEE Transactions on Evolutionary Computation*, 26(6):1539–1552, 2022.

David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

Saeed Hedayatian and Stefanos Nikolaidis. AutoQD: Automatic discovery of diverse behaviors with quality-diversity optimization. In *International Conference on Learning Representations (ICLR)*, 2026.

Joey Hong, David Dohan, Rishabh Singh, Charles Sutton, and Manzil Zaheer. Latent programmer: Discrete latent codes for program synthesis. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2021.

Joel Lehman and Kenneth O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19(2):189–223, 2011.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9493–9500, 2023.

Guan-Ting Liu, En-Pei Hu, Pu-Jen Cheng, Hung-yi Lee, and Shao-Hua Sun. Hierarchical programmatic reinforcement learning via learning to compose programs. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2023.

Max Liu, Chan-Hung Yu, Wei-Hsu Lee, Cheng-Wei Hung, Yen-Chun Chen, and Shao-Hua Sun. Synthesizing programmatic reinforcement learning policies with large language model guided search. In *International Conference on Learning Representations (ICLR)*, 2025. URL `https://arxiv.org/abs/2405.16450`.

Julian R. H. Mariño, Rubens O. Moraes, Tassiana C. Oliveira, Claudio Toledo, and Levi H. S. Lelis. Programmatic strategies for real-time strategy games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pp. 381–389, 2021.

Leandro C. Medeiros, David S. Aleixo, and Levi H. S. Lelis. What can we learn even from the weakest? Learning sketches for programmatic strategies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2022.

Vincent Micheli, Eloi Alonso, and François Fleuret. Transformers are sample-efficient world models. In *International Conference on Learning Representations (ICLR)*, 2023.

microRTS AI Competition. microRTS AI competition. `https://sites.google.com/site/micrortsaicompetition/introduction?authuser=0`. Official competition site and agent resources.

microRTS AI Competition. 2024 CoG competition results. `https://sites.google.com/site/micrortsaicompetition/competition-results/2024-cog-results`, 2024. Mayari placed first in the Classic track, all maps.

Rubens O. Moraes, Julian R. H. Mariño, Levi H. S. Lelis, and Mario A. Nascimento. Action abstractions for combinatorial multi-armed bandit tree search. In *Proceedings of the Fourteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, pp. 74–80, 2018.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

Suraj Nair, Aravind Rajeswaran, Vikash Kumar, Chelsea Finn, and Abhinav Gupta. R3M: A universal visual representation for robot manipulation. In *Proceedings of the Conference on Robot Learning (CoRL)*, 2022.

Andrew Y. Ng, Daishi Harada, and Stuart J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning (ICML)*, pp. 278–287. Morgan Kaufmann, 1999.

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

Santiago Ontañón, Nicolas A. Barriga, Cleyton R. Silva, Rubens O. Moraes, and Levi H. S. Lelis. The first MicroRTS artificial intelligence competition. *AI Magazine*, 39(1), 2018.

Wenjie Qiu and He Zhu. Programmatic reinforcement learning without oracles. In *International Conference on Learning Representations (ICLR)*, 2022. URL `https://openreview.net/forum?id=6Tk2noBdvxt`.

Pierre Sermanet, Corey Lynch, Yevgen Chebotar, Jasmine Hsu, Eric Jang, Stefan Schaal, and Sergey Levine. Time-contrastive networks: Self-supervised learning from video. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2018.

Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J. Lim. Learning to synthesize programs as interpretable and generalizable policies. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Constantinos Tsakonas and Konstantinos Chatzilygeroudis. Vector quantized-elites: Unsupervised and problem-agnostic quality-diversity optimization. *IEEE Transactions on Evolutionary Computation*, 2025.

Aäron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2018.

Abhinav Verma, Hoang M. Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Lijun Yu, Yong Cheng, Kihyuk Sohn, José Lezama, Han Zhang, Huiwen Chang, Alexander G. Hauptmann, Ming-Hsuan Yang, Yuan Hao, Irfan Essa, and Lu Jiang. MAGVIT: Masked generative video transformer. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.

Kevin Zakka, Andy Zeng, Pete Florence, Jonathan Tompson, Jeannette Bohg, and Debidatta Dwibedi. XIRL: Cross-embodiment inverse reinforcement learning. In *Proceedings of the Conference on Robot Learning (CoRL)*, 2021.

A Appendix

A.1 Bilevel Behavioural Search

This appendix gives the complete bilevel procedure summarized in Section 3.2, including its inner program search. We write $J(p)$ for $J(\pi_p)$, the return of the policy encoded by p in the current search interaction.

input: program space $\mathcal{P}$; behaviour abstraction space $\mathcal{V}$ induced by tokenizer Φ; outer CMAB search B (Naive Sampling); program library L; search budget
output: best program found
initialize B's local and global statistics, $L \leftarrow \emptyset$, $p^{\text{best}} \leftarrow$ NONE
while budget remains **do**
  $v^\star \leftarrow B.\text{PROPOSETARGET}()$
  if $L = \emptyset$ **then**
    $p_0 \leftarrow$ initial seed program
  else
    $p_0 \leftarrow \text{MOSTSIMILAR}(L, v^\star)$
  end if
  $(p', \text{evaluated}) \leftarrow \text{LOCALSEARCH}(p_0, v^\star)$ ▷ *Algorithm 2*
  for $(p'', v'', J'') \in \text{evaluated}$ **do**
    $B.\text{UPDATE}(v'', J'')$ ▷ *own observation, not* $v^\star$
    $L \leftarrow L \cup \{(p'', v'', J'')\}$
  end for
  if $p^{\text{best}} = \text{NONE}$ or $J(p') > J(p^{\text{best}})$ **then**
    $p^{\text{best}} \leftarrow p'$
  end if
end while
return p^{best}

Algorithm 1: Bilevel Behavioural Search with VIES.

Relation to VIES. The bilevel mechanism follows Aleixo & Lelis (2023). Their original behavioural space is defined through handcrafted domain-specific features. VIES preserves the separation between outer behavioural search and inner program search, but replaces those handcrafted features with a discrete behaviour abstraction induced from execution video. Each position of a VIES signature is a variable of the outer combinatorial search, and the possible codebook values provide the alternatives available for that variable. Candidate generation is unaffected by this substitution: at every step, candidate programs are still produced by the neighbourhood operator of the synthesizer in use (Section 5.1), whether that operator mutates a DSL program or queries a language model. VIES does not generate, decode, or mutate programs; the learned representation enters only through target construction, similarity, library retrieval, and the statistics of the outer combinatorial search.

Program library. The library stores every program evaluated during search together with its observed behavioural representation and the return obtained during that evaluation. When the outer search proposes a target $v^\star$, the library is queried for the stored program whose observed behaviour is most similar to $v^\star$, and that program initializes the inner local search. This lets the outer behavioural search redirect, or informally "jump," the inner search toward different regions of $\mathcal{P}$ without modifying the program representation or the neighbourhood operator; "jump" is used here only as an intuition, not as a formal operation.

Outer search: CMAB and Naive Sampling. As in Section 3.2, the outer search proposes $v^\star$ using Naive Sampling for combinatorial multi-armed bandits (Ontañón, 2017). Appendix A.2 gives the complete formulation – the CMAB mapping, its Local and Global MABs, and Algorithm 3, which implements $B.\text{PROPOSETARGET}()$ in Algorithm 1 below.

Each entry of L is an evaluation record (p, v, J), where v is the behavioural signature observed for a specific evaluation of p and J its return; L does not record a hypothetical deterministic mapping from a program to a single behaviour, since a program may induce different trajectories, and therefore different signatures, across evaluations (Section 4.2). Formally,

$$\text{MOSTSIMILAR}(L, v^\star) = p_0 \text{ such that } (p_0, \hat{v}, \hat{J}) \in \underset{(p,v,J)\in L}{\arg\max}\ \text{sim}(v, v^\star) :$$

the program component of the stored evaluation record whose observed behaviour is closest to the proposed target, used to initialize the inner search.

input: starting program p_0; behavioural target $v^\star$; neighbourhood operator; tokenizer Φ; annealing temperature schedule T and constant β; inner-search budget

execute p_0: obtain τ, $J(p)$, $v(\tau) \leftarrow \Phi(x(\tau))$
$p \leftarrow p_0$, $p^{\text{best}} \leftarrow p_0$, evaluated $\leftarrow \{(p_0, v(\tau), J(p))\}$
while inner-search budget remains **do**
  $p' \leftarrow \text{NEIGHBOUR}(p)$
  execute p'; obtain τ', $J(p')$, $v(\tau') \leftarrow \Phi(x(\tau'))$
  $G(\tau' \mid v^\star) \leftarrow \text{sim}(v(\tau'), v^\star)$
  evaluated $\leftarrow$ evaluated $\cup \{(p', v(\tau'), J(p'))\}$
  if $J(p')$, $J(p)$ differ beyond the comparison band **then**
    accept p' iff $J(p') > J(p)$ ▷ *return decides*
  else
    $\Delta G \leftarrow G(\tau' \mid v^\star) - G(\tau \mid v^\star)$
    accept p' with probability $\min\{1, \exp(\beta\, \Delta G/T)\}$ ▷ *simulated annealing*
  end if
  if p' accepted **then**
    $p \leftarrow p'$, $\tau \leftarrow \tau'$
  end if
  if $J(p') > J(p^{\text{best}})$ **then**
    $p^{\text{best}} \leftarrow p'$
  end if
end while
return p^{best}, evaluated

Algorithm 2: Local Search Guided by a Behavioural Target.

The update inside the **for** loop of Algorithm 1 associates return with the behavioural representation actually observed during evaluation rather than with the requested target: the statistics attached to a proposed target $v^\star$ are not updated with the returns of programs evaluated while pursuing it. Instead, every evaluated program updates the outer search using its own observed behavioural vector v'' and its own return J'', whatever target that evaluation happened to be guided by. The library is populated the same way, so both the outer search's statistics and its retrieval index reflect observed behaviour rather than requested targets.

The temperature T decreases over inner-search iterations following a fixed cooling schedule.

Return is always the primary acceptance criterion in Algorithm 2: outside the configured comparison band, acceptance is deterministic and requires a strict return improvement, and behavioural similarity never overrides a return difference outside that band. Inside the band, acceptance follows the simulated-annealing rule above, applied to the similarity difference ΔG rather than to return. Every candidate p' is still produced by the synthesizer's own neighbourhood operator; the tokenizer only maps the resulting trajectory to a signature and a similarity value.

Self-play versus best response. The two procedures above describe self-play, where the outer CMAB proposes behavioural targets directly in $\mathcal{V}$ and each target initializes and guides an inner program search (Algorithm 2). Best-response synthesis does not run the outer target-generation loop: $v^\star$ is fixed in advance, derived from the execution of a reference policy against the fixed opponent, and Algorithm 2 is run once, directly, with that fixed target as behavioural guidance.

A.2 Naive Sampling for Behavioural Targets

This appendix describes the operation abbreviated as $B.\text{PROPOSETARGET}()$ in Algorithm 1: how the outer combinatorial search constructs a behavioural target $v^\star$. The mechanism is Naive Sampling for combinatorial multi-armed bandits (CMABs) (Ontañón, 2017); Aleixo & Lelis (2023) use it to search a handcrafted behavioural space inside Bi-S, and VIES reuses the same mechanism over its learned behaviour abstraction space $\mathcal{V}$ instead.

CMAB formulation. Following Ontañón (2017), a CMAB is defined over a set of variables $X = \{X_1, \ldots, X_n\}$, where variable X_i has arms $X_i = \{v_i^1, \ldots, v_i^{K_i}\}$. The space of complete combinations is $\mathcal{X} = X_1 \times \cdots \times X_n$, and a complete assignment $V = (v_1, \ldots, v_n) \in \mathcal{X}$ is a *macro-arm*. In the VIES instantiation, $n = R$, the number of positions in a VIES signature, and every variable shares the same codebook, so $X_r = \{1, \ldots, K\}$ for $r = 1, \ldots, R$; the resulting $\mathcal{X}$ is exactly the behaviour abstraction space $\mathcal{V}$. Each VIES signature position is therefore one CMAB variable X_r, each codebook value is one arm of that variable, and a complete selection of code values is a macro-arm; the macro-arm proposed by Naive Sampling is exactly the behavioural target $v^\star = (v_1^\star, \ldots, v_R^\star) \in \mathcal{V}$. "Macro-arm" is Naive Sampling's term for this object; $v^\star$ is the corresponding object in the VIES formulation.

The naive assumption. Evaluating every macro-arm directly would require exploring all K^R combinations. Ontañón (2017) avoid this with the *naive assumption*: the utility of a macro-arm $V = (v_1, \ldots, v_n)$ is approximated through the contributions of its individual variable-value assignments,

$$\mu(V) \approx \sum_{i=1}^{n} \mu_i(v_i).$$

This lets information from one evaluated macro-arm inform future macro-arms that reuse some of its variable values, without evaluating all complete combinations independently. We do not claim that the true utility of a VIES behavioural target is actually additive across signature positions; the naive assumption is a sampling approximation that guides exploration of $\mathcal{V}$, not a modelling claim about it.

Local MABs. For each variable X_r, Naive Sampling defines a *Local MAB*, MAB_r, whose arms are the K codebook values at position r. Following Ontañón (2017)'s notation, adapted here to index by signature position r rather than the generic i, let $T_r^t(v_r^k)$ denote the number of times value k has been selected for X_r up to iteration t, and $\mu_r^t(v_r^k)$ the corresponding marginalized average reward. Consistent with Algorithm 1, each evaluated program contributes its return to the variable-value statistics corresponding to the code values actually observed in its behavioural signature v'', rather than to the values requested in $v^\star$: μ_r^t is updated from this observed vector and the program's own return, never from $v^\star$ directly.

Global MAB. The *Global MAB*, MAB_g, treats macro-arms as its arms rather than individual values, maintaining the analogous pair of statistics keyed by complete macro-arms. MAB_g initially contains no arms. In our instantiation, MAB_g gains a new arm, or updates an existing one, when the outer update in Algorithm 1 processes an evaluated program: the arm is keyed by that program's *observed* behavioural signature v'', not by whichever target $v^\star$ the inner search happened to pursue. A macro-arm constructed by the Local MABs during exploration (Algorithm 3) therefore does not itself become an arm of MAB_g at proposal time; it becomes one only if, and once, some evaluated program is observed to realize it. Following Ontañón (2017)'s intuition, the Local MABs are used to explore promising new macro-arms, while the Global MAB is used to exploit macro-arms that have already produced promising results.

Policies π_0, π_l, π_g. Ontañón (2017) name three policies: π_0 determines whether Naive Sampling explores through the Local MABs or exploits through the Global MAB; π_l selects a value from a Local MAB during exploration; π_g selects a previously sampled macro-arm from MAB_g during exploitation.

Exploration. If π_0 selects exploration, Naive Sampling constructs a new macro-arm: for each variable X_r, it applies π_l to MAB_r and selects a value for each variable independently, $v_r^\star \in X_r$; the R selections are combined into $v^\star = (v_1^\star, \ldots, v_R^\star)$, which is returned as the behavioural target used by Algorithm 1. This macro-arm becomes an arm of MAB_g only later, through the outer update described under *Global MAB* above, and only if it is the signature some program is actually observed to realize.

Exploitation. If π_0 selects exploitation, Naive Sampling instead applies π_g over MAB_g and selects one of the macro-arms already present there as the next $v^\star$, without constructing new coordinate values.

Algorithm 3 implements $v^\star \leftarrow B.\textsc{ProposeTarget}()$ in Algorithm 1. The resulting macro-arm then flows through the rest of the bilevel loop: Naive Sampling proposes $v^\star$; the library is queried for

input: Local MABs $MAB_1, \ldots, MAB_R$; Global MAB MAB_g; policies π_0, π_l, π_g
output: macro-arm / behavioural target $v^\star$

mode $\leftarrow \pi_0()$
if mode $=$ explore **then**
  for $r = 1, \ldots, R$ **do**
    $v_r^\star \leftarrow \pi_l(MAB_r)$ ▷ *select independently*
  end for
  $v^\star \leftarrow (v_1^\star, \ldots, v_R^\star)$
  else
  $v^\star \leftarrow \pi_g(MAB_g)$
  end if
return $v^\star$

Algorithm 3: Naive Sampling Target Selection (B.PROPOSETARGET()).

the stored evaluation record whose observed behaviour is closest to it; the inner search (Algorithm 2) explores program space from that starting point; and the behavioural signatures and returns it actually observes – not $v^\star$ itself – are what update the Local MABs, the Global MAB, and the library, as described under *Global MAB* above. Algorithm 3 never generates, mutates, or evaluates a program; it operates only over macro-arms in $\mathcal{V}$.

Epsilon-greedy and two-phase configuration. Ontañón (2017) instantiate π_0, π_l, π_g as epsilon-greedy policies, writing the configuration $NS(\epsilon_0, \epsilon_l, \epsilon_g)$: ϵ_0 is the probability that π_0 selects exploration (and $1 - \epsilon_0$ the probability of exploitation), ϵ_l is the epsilon parameter of π_l, and ϵ_g is the epsilon parameter of π_g; each epsilon-greedy policy selects its currently preferred arm with probability $1 - \epsilon$ and a random arm with probability ϵ. Ontañón also define *Two-Phase Naive Sampling*, $NS(k, \epsilon_0^1, \epsilon_l^1, \epsilon_g^1, \epsilon_0^2, \epsilon_l^2, \epsilon_g^2)$: the first k sampling iterations use one parameter set and the remainder use a second, letting a more exploratory phase precede a more exploitative one. In our implementation, π_0, π_l, π_g are epsilon-greedy and switch to a second set of parameter values partway through search, in the sense of Two-Phase Naive Sampling above; this configuration only changes how $v^\star$ is proposed and leaves the outer-search update in Algorithm 1 unchanged, which still updates from a program's own observed behavioural vector and return, never from the requested target.

Why this works for VIES. $\mathcal{V} = \{1, \ldots, K\}^R$ contains K^R points, far more than the outer search could enumerate. Discreteness gives a finite set of choices at each signature position, and factorization gives one variable per position: codebook values become the arms of the corresponding Local MAB, and complete signatures become macro-arms. Through the naive assumption, the Local MABs' marginalized statistics are reused across different macro-arms, so information from one evaluation informs many future proposals, while the Global MAB lets the search exploit complete macro-arms already sampled. The discrete VIES representation therefore supplies the factorized structure required by the CMAB outer search, and Naive Sampling can consequently search $\mathcal{V}$ without enumerating all K^R combinations. As Section 4.3 already notes, not every point of $\mathcal{V}$ need correspond to a behaviour some program can realize.

A.3 GUIDANCE CONDITION DETAILS

Handcrafted representation in MicroRTS. The handcrafted behavioural representation consists of the production counts introduced by Medeiros et al. (2022) and later used by Aleixo & Lelis (2023): the numbers of workers, light, ranged, and heavy units, bases and barracks built, and resources returned.

Handcrafted representation in Can't Stop. The handcrafted behavioural representation is the one proposed by Medeiros et al. (2022), which records the final progress reached by the player in each board column across repeated games; Appendix A.4 describes how it is accumulated and compared.

Raw video. Raw video compares the candidate and reference execution recordings directly using mean pixelwise L_1 distance, without discretization. When the recordings have different lengths, the shorter recording is padded with black frames to match the longer one before comparison; this padding is specific to the comparison and differs from the tokenizer preprocessing of Appendix A.6.

A.4 CAN'T STOP SELF-PLAY EVALUATION PROTOCOL

In Can't Stop, repeated executions of the same program can produce different trajectories because of stochastic dice outcomes. We therefore do not associate a candidate program with a single VIES signature $v(\tau)$ taken from one execution. Instead, behavioural observations from multiple executions are accumulated into an aggregate behavioural representation, and it is this aggregate representation, not an individual $v(\tau)$, that search actually compares.

Program evaluation (level 1). A candidate is evaluated by playing 100 games against every opponent currently in the accumulated opponent pool (Section 4.4); its return is the resulting win rate (win $= 1$, draw $= 0.5$, loss $= 0$), pooled across all of these games. If the pool contains M opponents, level 1 is therefore estimated from $100M$ games in total. As in Appendix A.1, this pooled win rate is what is compared against the configured comparison band: outside the band it alone determines acceptance, and inside the band – when two candidates' win-rate estimates are statistically indistinguishable given their sampling uncertainty, $|J_a - J_b| \leq \sqrt{m_a^2 + m_b^2}$ for uncertainty margins m_a, m_b – the level-2 behavioural signal described next then governs the simulated-annealing acceptance rule instead.

Behavioural representation (level 2). The behavioural representation is built only from 30 of the games played against the most recently added opponent in the pool, not from the full opponent pool used for level 1. Following the player-perspective convention of Section 4.4, only the trajectory in which the candidate plays as player 0 enters this representation; in Can't Stop this is automatic, since the candidate is always seated in the same fixed seat, so every recorded execution is already from its own perspective. For each of these 30 executions, the tokenizer produces a VIES signature $v(\tau) = (v_1, \ldots, v_R)$, with $v_r \in \{1, \ldots, K\}$ ($R = 24$, $K = 10$ for Can't Stop; Appendix A.5). Rather than tokenizing one trajectory into one signature and comparing it directly, the 30 selected executions are combined into a single *behavioural table* $A \in \mathbb{N}^{R \times K}$, indexed by signature position r and codebook value k: the entry $A_{r,k} = \#\{m : v_r(\tau_m) = k\}$ counts how many of the 30 executions take value k at position r. Rows correspond to VIES latent positions and columns to codebook values; the table accumulates raw observation counts, retaining the empirical variation produced by the stochastic executions instead of collapsing them into a single deterministic signature.

Behavioural target centroid. The program library and the outer CMAB retain behavioural information at different granularities. A library entry associates a program with the aggregate behavioural table constructed from its 30 selected executions against the most recently added opponent. The CMAB, by contrast, retains behavioural vectors individually. To construct a behavioural target, five individual vectors $v^{\star(1)}, \ldots, v^{\star(5)}$ are sampled from the CMAB, each $v^{\star(j)} = (v_1^{\star(j)}, \ldots, v_R^{\star(j)}) \in \mathcal{V}$; each contributes one observation at each latent position. These five vectors are accumulated into a target count table $C^\star \in \mathbb{N}^{R \times K}$, of the same form as the candidate table A defined above:

$$C^\star_{r,k} = \sum_{j=1}^{5} \mathbf{1}[v_r^{\star(j)} = k].$$

Its normalized frequencies $\bar{C}^\star = C^\star/5$ define the behavioural target centroid used by the inner search: the level-2 comparison is the table similarity, defined next, between the candidate's own table A and this target table $C^\star$.

Table similarity. The handcrafted and VIES representations described below are compared using the same rule. Let A and B be two behavioural tables, accumulated over n_A and n_B executions respectively, and let $\bar{A} = A/n_A$ and $\bar{B} = B/n_B$ be their normalized frequency tables. Define the valid cells as $\mathcal{I} = \{(r, c) : A_{r,c} + B_{r,c} > 0\}$, so a cell is ignored only if neither table observed it. For every valid cell, let $d_{r,c} = |\bar{A}_{r,c} - \bar{B}_{r,c}|$ and weight it by the combined raw count $w_{r,c} = A_{r,c} + B_{r,c}$.

Similarity is then one minus a weighted mean absolute difference between the normalized tables:

$$\mathrm{sim}(A, B) = 1 - \frac{\sum_{(r,c) \in \mathcal{I}} w_{r,c}\, d_{r,c}}{\sum_{(r,c) \in \mathcal{I}} w_{r,c}}.$$

Cells absent from both tables are ignored and are not treated as zero similarity; a cell observed by only one of the two tables remains valid and contributes its full normalized value as the difference, weighted by whatever count that one table assigned to it; and more frequently observed cells receive more weight in the estimate.

Handcrafted behavioural representation. Following Medeiros et al. (2022), the handcrafted condition for Can't Stop accumulates a table of the same form but over a different quantity. For each evaluated game, only the final board state is used, and only the entries corresponding to the candidate's own player are read from it; for each of the eleven board columns, the highest row reached by that player is identified, when such a marker is present, and the corresponding entry of a 14×11 count table is incremented. Accumulated over repeated games, this table is a histogram of where the player tends to finish in each column, rather than a video-induced signature.

Comparing handcrafted and VIES conditions. The handcrafted and VIES conditions differ only in which behavioural table is accumulated, not in how that table is aggregated or compared: the handcrafted representation records final progress by board column, in a 14×11 table, whereas VIES records codebook assignments at the R learned latent positions, in an $R \times K$ table. These two tables differ both in dimensionality and in the information they retain about an execution; they are accumulated over the same kind of repeated stochastic executions and compared with the identical normalized, count-weighted absolute-difference rule above, which makes the behavioural representation the principal experimental difference between the two guidance conditions.

Level 1 and level 2 together. Level 1 (pooled win rate) and level 2 (table similarity) are never averaged into a single score. Level 1 determines the outcome whenever the comparison band leaves no statistical ambiguity between two candidates; only when they are tied at level 1 does the aggregate behavioural similarity between candidate and target tables supply the auxiliary signal that guides the search, exactly as described for the general bilevel procedure in Appendix A.1.

A.5 Can't Stop Tokenizer Training

Training corpus. The video tokenizer used for Can't Stop is trained on a corpus generated independently of the programs synthesized in our search experiments (Section 5.1). The corpus is produced by three fixed players available in the implementation: the Glenn heuristic player (Glenn & Aloi, 2009), a random baseline, and a rule-based heuristic. These are simply the three fixed players available for corpus generation, not a selection intended to maximize behavioural diversity. For continue-or-stop decisions, the random baseline selects either option with probability 0.5; when choosing a column, it samples uniformly from the available actions. The rule-based heuristic instead follows explicit risk-sensitive rules: it favours columns that are easier to roll, columns where progress has already been made, and columns closer to completion, and it stops rolling once continuing becomes sufficiently risky. These three players are paired in all $3 \times 3 = 9$ ordered matchups, including self-play against themselves, and 100 games are generated per matchup, yielding a training corpus of $9 \times 100 = 900$ trajectories – independent of, and disjoint from, the games played during search-time evaluation described in Appendix A.4.

Trajectory representation. Each game trajectory is stored as a tensor of shape $(T, 3, 13, 11)$, with one frame recorded at every dice-roll step together with the game's final state. The three channels represent one player, the neutral markers, and the other player, following the board encoding used throughout this work.

Preprocessing. Trajectories are converted into fixed-length windows of 128 frames with stride 32; trajectories shorter than 128 frames are zero-padded to that length. These windows form the tokenizer's training inputs.

Tokenizer dimensions. The Can't Stop encoder reduces each input window to a latent grid of $6 \times 2 \times 2$, giving $R = 24$ discrete token positions, each quantized against a codebook of size $K = 10$ using cosine similarity; the codebook itself is updated through an exponential moving average rather than by gradient descent.

Training objective. The tokenizer is trained with a reconstruction objective together with a vector-quantization commitment term on the encoder output; the codebook is updated by the exponential-moving-average rule above rather than by this loss. A masked-token prediction loss is computed and monitored throughout training but does not contribute to the optimization objective in the current Can't Stop training implementation.

Role of the tokenizer during synthesis. During synthesis, VIES uses only the trained encoder and vector quantizer to obtain the discrete code indices of an execution; the decoder and the masked-token predictor are not required to compute behavioural signatures.

A.6 MicroRTS Tokenizer Training

Training corpus. The MicroRTS tokenizer corpus is generated independently of the programs synthesized in our search experiments (Section 5.1). It is produced by round-robin matches among a fixed roster of existing MicroRTS agents, including agents obtained from official microRTS AI Competition resources (microRTS AI Competition) and standard baseline agents available in the implementation; the roster is simply the set of existing agents available for corpus generation, not a selection intended to maximize behavioural diversity. This round-robin generation produced 3,169, 3,254, and 2,745 recordings for the `workers_24x24`, `basesWorkers32x32A`, and `CombatMap` configurations respectively, before preprocessing and opponent filtering; these are not the final training-set sizes. Recordings involving MayariBot are excluded before tokenizer training, so the fixed opponent used in our best-response experiments (Section 5.1) is unseen during representation training; MayariBot was present in the tournament roster and is filtered only from the recordings used to train the tokenizer. This corpus-generation process is independent of, and distinct from, the protocol used elsewhere in our experiments to evaluate synthesized programs against fixed opponents.

Trajectory representation and preprocessing. Each MicroRTS match is represented as a three-channel video. Consecutive duplicate frames are first removed, so that every retained frame differs from the preceding one. Up to 128 of the remaining frames are then retained: longer sequences are cropped to this 128-frame temporal window, so events outside it cannot affect the resulting signature, and sequences with fewer than 128 frames are padded with black frames. The resulting tensor, of shape $(3, 128, H, W)$ with values scaled to $[0, 1]$, matching the decoder's Sigmoid output, is then passed to the tokenizer. We split recordings 80/20 by game using a fixed seed of 42, so no recording appears in both partitions.

Tokenizer architecture. The MicroRTS tokenizer follows the general MAGVIT-style design described in Section 3.3. A 3D convolutional encoder, using stacked 3D convolutions with normalization and nonlinearities, reduces the 128-frame input to a compact spatiotemporal latent grid. Each latent vector is discretized by a cosine-similarity vector quantizer: the latent vector and the codebook embeddings are normalized, and the code with maximum cosine similarity is selected. The codebook, of size $K = 20$, is updated through an exponential moving average (decay 0.99) rather than by gradient descent, and the encoder path uses a straight-through estimator. A 3D decoder, using upsampling and residual blocks, reconstructs the input from the quantized latent grid, ending in a Sigmoid output. A masked transformer (depth 4, 4 attention heads, dropout 0.1) additionally predicts randomly hidden code indices from their surrounding context, with 25% of token positions masked during training.

Map-specific latent dimensions. Table 1 reports, for each map, the tokenizer input shape, the resulting latent grid, the number R of VIES signature positions, and the training batch size. The leading dimension of each latent shape is the latent embedding size and is not part of R; R is the product of the remaining temporal, height, and width dimensions. Together with $K = 20$, these give the behaviour abstraction spaces $\mathcal{V} = \{1, \ldots, 20\}^{36}$, $\{1, \ldots, 20\}^{64}$, and $\{1, \ldots, 20\}^{32}$ used for `workers_24x24`, `basesWorkers32x32A`, and `CombatMap` respectively (Section 4.2).

`CombatMap`'s tokenizer input is stored as an 8×16 (height×width) tensor; this is the same map described as 16×8 in Section 5.1, under the opposite width×height convention.

Table 1: MicroRTS tokenizer configuration per map. R is the number of VIES signature positions; all three configurations use codebook size $K = 20$.

Map	Input shape	Latent shape	R	Batch size
`workers_24x24`	$(3, 128, 24, 24)$	$(32, 4, 3, 3)$	36	192
`basesWorkers32x32A`	$(3, 128, 32, 32)$	$(32, 4, 4, 4)$	64	128
`CombatMap`	$(3, 128, 8, 16)$	$(32, 4, 2, 4)$	32	192

Training objective. The MicroRTS tokenizer is trained with

$$\mathcal{L} = \mathcal{L}_{\text{recon}} + \mathcal{L}_{\text{vq}} + 0.1\,\mathcal{L}_{\text{CE}} + 10\,\mathcal{L}_{\text{focal}}.$$

$\mathcal{L}_{\text{recon}} = \text{MSE}(\hat{x}, x)$ is the mean squared reconstruction error. $\mathcal{L}_{\text{vq}}$ is a commitment term, the mean squared error between the encoder output and its detached quantized representation; the direct codebook loss is disabled, since the codebook is instead updated by the exponential-moving-average rule described above. $\mathcal{L}_{\text{CE}}$ is the cross-entropy of the masked transformer's predictions, evaluated only at masked token positions; unlike in Can't Stop (Appendix A.5), this masked-token loss contributes directly to the MicroRTS optimization objective. $\mathcal{L}_{\text{focal}}$ reweights the reconstruction error toward occupied board cells,

$$\mathcal{L}_{\text{focal}} = \text{mean}\big[\,|\hat{x} - x|^2\,(1 + 6 \cdot \mathbf{1}[x > 0])\big]\,,$$

so that pixels containing units or resources receive six times the local error weight of background pixels within this term. MicroRTS frames contain large background regions, and this term gives greater reconstruction weight to occupied, non-zero pixels; the final objective weights this term by 30.

Optimization. Training uses AdamW with learning rate 10^{-4} and no weight decay, with gradients clipped to a maximum norm of 1.0. Checkpoints are saved every 10 epochs, and the checkpoint with the lowest validation loss is retained.

Role of the tokenizer during synthesis. During tokenizer training, the encoder, vector quantizer, decoder, and masked transformer participate in the same training pipeline, with the codebook updated by EMA. During program synthesis, only the trained encoder and vector quantizer are used to obtain the discrete code assignments that form a VIES signature; the decoder and the masked transformer are not required to encode a completed execution into its signature.

A.7 LLM-Based Synthesis Details

Prompt construction. In the LLM-based MicroRTS experiments, the language model acts as the mutation operator of the local search. For the gpt-oss-120b experiments reported in Figure 2(a), the system prompt provides the game rules, engine constraints, available API and abstraction layer, information available about the opponent, and implementation patterns intended to prevent invalid agents. The model has no access to the remaining codebase. At each search step, the user message contains the complete current program together with evaluation feedback and requests one deliberate modification. The model is instructed to modify one mechanism rather than rewrite the program for style, preserve the `GeneratedAgent` interface, and return the complete candidate as a single Python code block.

Evaluation feedback. The feedback always reports the candidate's task score and its win/draw/loss counts, together with an error message when evaluation fails. When behavioural guidance is enabled, the feedback additionally includes the behavioural guidance signal.

Candidate output and failures. Each invocation is requested to produce one complete Python implementation of `GeneratedAgent`. The output must contain a single Python code block and no additional prose. When an evaluated program fails, the next feedback includes the observed error so that subsequent mutations can address the implementation failure.

Decoding configuration. Both gpt-oss-120b and gemini-2.5-flash-lite use a temperature of 0.6, set through the command-line `--temperature` argument and therefore overriding the client-class default of 0.1. For both models, `top_p` is 0.95; this parameter is not exposed through the command line and therefore remains at the default value configured by each model client.

A.8 Search Hyperparameters

Budgets and restarts. In MicroRTS best-response synthesis, DSL-based search is restarted every 40 minutes, and LLM-based search is restarted after 40 consecutive steps without improvement. In MicroRTS self-play, each inner search is limited to 40 minutes and uses four CPUs in parallel, with a random seed sampled separately for each CPU. In Can't Stop self-play, each inner search is limited to 7,000 evaluated programs and uses eight CPUs only to parallelize match execution; a single random seed is sampled for the search.

Naive Sampling. In MicroRTS, the outer combinatorial search uses two parameter phases. The first phase uses

$$(\epsilon_0, \epsilon_l, \epsilon_g) = (0.5, 0.7, 0.7),$$

and the second uses

$$(\epsilon_0, \epsilon_l, \epsilon_g) = (0.5, 0.9, 0.9).$$

For DSL-based MicroRTS synthesis, the transition from the first to the second phase occurs after 24 hours of wall-clock search time. In Can't Stop, Naive Sampling uses a single configuration throughout search,

$$(\epsilon_0, \epsilon_l, \epsilon_g) = (0.5, 0.7, 0.7),$$

with no second phase. The discount factor is 0.8.

Inner-search annealing. The behavioural component of the inner search uses the simulated-annealing acceptance rule described in Algorithm 2. Both domains use the same cooling form,

$$T_i = \frac{T_0}{1 + \alpha i},$$

where i is the current search iteration. For MicroRTS, $T_0 = 2000$, $\alpha = 0.9$, and $\beta = 0.5$ for both DSL- and LLM-based synthesis. There is no explicit minimum-temperature stopping criterion; search terminates according to its wall-clock or search-specific stopping condition rather than when T reaches a prescribed value.

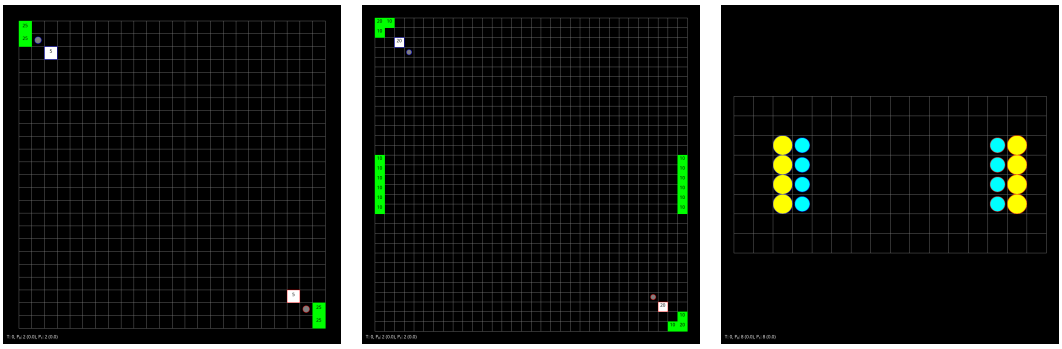
For Can't Stop, $T_0 = 100$, $\alpha = 0.6$, and $\beta = 100.0$. The inner search continues while $T > 1$, giving an effective final temperature of approximately 1.

B Complete LLM Prompt

For reproducibility, we report the prompting setup used for the `gpt-oss-120b` experiments in Figure 2(a). Each model invocation consists of a fixed system message (Appendix C.1) and a user message instantiated for the current search iteration from the candidate-mutation template (Appendix C.2), which embeds the current candidate program, its evaluation feedback (Appendix C.4), and a mutation instruction (Appendix C.3). Text shown in `<PLACEHOLDER>` form below is filled in at runtime for the current search step; it is not sent to the model literally. The reported experiments did not provide production-count profiles to the language model. We therefore reproduce below, verbatim from their source files, the fixed system prompt, the candidate-mutation template, and the dynamic components that were supplied during search.

C MicroRTS Test Maps

Figure 6 shows the initial state of each of the three MicroRTS maps used in our experiments: `workers_24x24` and `basesWorkers32x32A` support economic and combat behaviour through workers, bases, and resources, while `CombatMap` contains only four ranged and four heavy units per player, with no resources or structures.



(a) `workers_24x24` (b) `basesWorkers32x32A` (c) `CombatMap`

Figure 6: Initial state of the three MicroRTS maps used in our experiments.

C.1 System Prompt

The fixed system message supplied to the model at every search iteration. This listing runs to more than a thousand lines; a breakable colour box of the kind used below exceeds TEX's listing-frame memory budget at this length, so it is set in the same colour as a plain, page-breaking framed listing instead.

System Prompt

```
You write agents that play MicroRTS in Python. The documents below are
everything you have: the rules of the game, the contract the engine enforces and
the rules you must not break, the API you may call, the abstraction layer you
should prefer, and what is knowable about your opponent. There is no other
documentation, no access to the codebase, no information about who you are playing
against, and no advice on strategy --- what to build and when is yours to work out.


You write agents that play MicroRTS in Python.

# The game

MicroRTS is a fully observable, DETERMINISTIC, grid-based real-time strategy
game. Two players own units on a rectangular grid. You never change the world
directly: each decision cycle you propose a set of unit actions, and the engine
validates them, executes them over several cycles, and advances the clock.

A player wins by destroying every enemy unit. There is no other win condition:
holding more territory or more resources decides nothing on its own.

# World model

The map is a fixed rectangular grid, addressed by (x, y) with x in
[0, width) and y in [0, height). Each cell holds at most one unit, and may
instead be a terrain wall. Movement and every interaction are limited to the 4
cardinal directions; there are no diagonals.

Grid convention, which is easy to get backwards:

    UP    is y - 1        DOWN  is y + 1
    LEFT  is x - 1        RIGHT is x + 1

So y grows downwards. A cell is unusable if a unit stands on it, if it is a
wall, OR if some in-progress action has reserved it (a unit moving there, or a
structure being produced there). Reservation is invisible in the unit lists ---
ask the engine with gs.free(x, y) instead of scanning units yourself.

# Units

Every unit has a position, an owner, a type, hit points, and possibly carried
resources. Neutral Resource units belong to no player (owner id -1).

  type      cost  hp  damage  range  produceTime  moveTime  attackTime  notes
  Base        10  10     -      -        200         -          -       stockpile; produces
     Workers; cannot move or attack
```

```
  Barracks     5   4     -      -        100         -          -       produces Light/Heavy/
     Ranged; cannot move or attack
  Worker       1   1     1      1         50         10          5      harvests and builds
  Light        2   4     2      1         80          8          5      melee
  Heavy        3   8     4      1        120         10          5      melee
  Ranged       2   1     1      3        100         10          5      attacks from range
  Resource     -   1     -      -          -         -          -       neutral, immobile,
     harvestable

Those are the numbers of the unit type table this project uses
(UnitTypeTable(2)). Read them from the objects where you can ---
unit.getType().getCost(), unit.getMaxHitPoints(), unit.getAttackRange() --- so a
different table does not silently break your agent.

The table above is game DATA --- costs, hit points, damage, timings. It says
nothing about which unit is better against which, when to build what, or what
composition wins: that is the part you have to work out, and the score is the
only thing that will tell you whether you were right.

# Economy

Resources are one shared pool per player, spent on production. A Worker walks to
a neutral Resource unit, harvests 1 (harvestTime 20), carries it back to a Base
and returns it (only then does the pool grow). Resource units deplete and
disappear. Both players start with 5 resources on the standard maps.

# Production

The hierarchy is strict, and nothing else can produce:

    Base     -> Worker
    Worker   -> Base, Barracks
    Barracks -> Light, Heavy, Ranged

Production always goes into one of the 4 cells adjacent to the producer, and
that cell must be free. The producer is busy for the whole produceTime, so
production capacity is limited by how many producing structures you own, not
only by resources.

# Combat

An attack targets a COORDINATE, not a unit. If an enemy occupies that
coordinate when the attack resolves, it takes damage; if it moved away, the
attack hits nothing. Attacks are legal only within the unit's attack range,
measured as squared Euclidean distance:

    (ux - x)^2 + (uy - y)^2 <= range^2

A unit cannot do anything else while attacking.

# Time and decisions

Time advances in discrete cycles. Every action takes several cycles, and while a
unit is executing one it is BUSY: it cannot accept another, and the action
cannot be cancelled or replaced. You may therefore only give an action to an
IDLE unit. Your getActions method is called every cycle, so most calls have
little to do --- the units you commanded are still busy carrying it out.


# The engine contract

The loop is always the same, and any logic that bypasses it is invalid:

    observe the GameState  ->  propose a PlayerAction  ->  the engine validates
    and executes it  ->  the engine advances time

A PlayerAction is a set of (Unit, UnitAction) pairs. It is accepted only if:
- every unit in it is yours and is idle;
- no unit appears twice;
- no two actions want the same cell;
- the TOTAL resource cost of the set fits your current pool.

Anything the engine considers invalid is silently discarded. You get no
exception and no message --- the unit simply does nothing, which is why a broken
agent looks like a passive one.

## Resources are budgeted per decision cycle, not per action

The engine validates the cost of the whole PlayerAction at once. Two productions
that are each affordable alone are rejected together when their sum exceeds your
pool. Two ways to stay consistent:
```

```
- keep one running budget yourself, starting from
  FuctionsUtil.availableResources(gs, player) --- the pool MINUS what actions
  already in flight have committed --- and subtract each cost as you add a
  production;
- or let the abstraction layer do it: translateActions() drops actions that no
  longer fit (see 04_abstraction.txt).

Note that gs.getPlayer(player).getResources() is the raw pool and does NOT
subtract what in-flight actions already committed. Budget with
FuctionsUtil.availableResources instead.

## Spatial exclusivity

A cell may be occupied by a unit or reserved by an in-progress action. You must
not move or produce into either. gs.free(x, y) covers both, but it knows nothing
about the actions you are still assembling this cycle: if you send two units to
the same free cell, gs.free says yes to both and the engine drops one. Track the
cells you have claimed this cycle in your own set.

# Hard rules --- breaking these forfeits the match

1. NEVER pass a negative coordinate to gs.free(x, y). The engine does not
   bounds-check it: the process dies with a SEGMENTATION FAULT, no exception, no
   traceback, and the match is lost. Read pgs.getWidth() / pgs.getHeight() once
   and guard EVERY coordinate you compute:

       if 0 <= x < width and 0 <= y < height and gs.free(x, y): ...

   A unit standing on the edge of the map always produces at least one
   out-of-range neighbour, so unguarded neighbour loops are not an edge case ---
   they are the single most common way a generated agent scores zero. The same
   guard belongs on pgs.getTerrain(x, y), which reads out of bounds silently and
   returns garbage.

2. Treat the GameState as immutable. Do NOT call gs.cycle(), gs.issueSafe(),
   gs.updateScream(), any setter (setX, setY, setHitPoints, setResources,
   setType), and do not build a GameState to look ahead. There is no simulator
   available to you: write a policy, not a search.

   Do not construct engine objects at all --- no Unit(...), no PhysicalGameState(...),
   no UnitType. Their constructors are C++ signatures and guessing the arguments
   raises TypeError on the first cycle. Represent positions as plain integers or
   tuples. PlayerAction() is the single object you are meant to create.

3. Command only your own units --- iterate pgs.getUnits(player). Never assign an
   action to an enemy unit (getUnits(1 - player)) or to a neutral Resource unit
   (getUnits(-1)).

4. Command only IDLE units: gs.getActionAssignment(unit) is None. At most one
   action per unit per cycle.

5. Validate before adding: gs.isUnitActionAllowed(unit, action), or take the
   action straight out of unit.getUnitActions(gs), which only ever contains
   legal actions.

6. Be fast and never block. getActions runs every cycle of a match that can last
   3000 cycles; a match is killed and scored as a loss if it exceeds its
   wall-clock limit. No sleeping, no unbounded loops, no I/O, no printing.

7. Import only the Python standard library plus the project names listed in
   03_api.txt and 04_abstraction.txt. No numpy, no torch, no network.

8. `gs` and `player` are ARGUMENTS, not global or stored state. A helper method
   that needs them must receive them: `def _rule_attack(self, gs, player, world)`.
   A helper that mentions `gs` without taking it raises NameError on the first
   cycle and loses the match --- this is one of the most common failures observed.
   Storing them on `self` at the top of getActions (`self.gs = gs`) is fine, but
   then use `self.gs` everywhere, consistently.

9. The file you output must be INTERNALLY CONSISTENT. You are writing the whole
   file, so a rename is only half done until every use is updated: if you change
   a method's signature, fix every call site; if you rename a dictionary key
   ("Worker" -> "Workers"), fix every read; if you drop a helper, remove its
   calls. TypeError ("takes 2 positional arguments but 3 were given") and
   KeyError are the usual symptoms, and both forfeit every match. Re-read your
   own output once before answering, checking that every name you call is defined
   with the arity you call it with.

# How your agent is scored
```

```
YOU ARE ALWAYS PLAYER 0. The opponent is always player 1. The `player` argument
of getActions will be 0 in every match --- but read it and use it anyway rather
than hard-coding 0, so the same code keeps working if that ever changes.

One match per map decides your score: win 1, draw 0.5, loss 0, and a match that
reaches the cycle cap is a draw. The match is replayed from the same starting
state and the same random state every time, so the same program always gets the
same score --- but with three possible values it is a blunt instrument, which is why
the finer objective below exists.

Being player 0 is the favourable side: the engine issues player 0's actions first,
so you win ties over contested cells.

When the score cannot separate two versions --- which is most of the time, since it
only takes three values --- a second, finer objective decides. Each match records
what your agent produced and how it played, and that is compared against a strong
reference agent on the same map, giving a similarity between 0 and 1. Among
versions with the same score, the more similar one is treated as better. You are
told both numbers for the program you are improving.

Read that similarity with care, because it can be raised by playing WORSE. It
rewards resembling the reference, including on the things the reference did not
do, and a match that ends early resembles a reference match less through what
happened than through what stopped happening. Two measured examples from real
searches, both of which the similarity scored ABOVE a working agent: one that
harvested for the entire match and never built a Barracks, and one that built
twelve Barracks, ten workers and not a single soldier.

So do not chase the number. The only reliable way to raise it is to play well:
keep the economy running, produce an army, and use it. If a change makes the
similarity go up while the agent visibly does less, it is a worse agent that
happens to measure better, and the score will eventually say so.

These count as losses: code that raises, code that hangs, code that segfaults.
Robustness is not a side concern here, it is most of the score.


# API --- the engine objects

Everything below was verified against the compiled module (MicroRTS_NB, a
nanobind binding of the C++ engine). Use ONLY these members, and only through
the getters --- never read or write attributes directly.

Available imports:

    from MicroRTS_NB import GameState, PhysicalGameState, Player, PlayerAction
    from MicroRTS_NB import ResourceUsage, Unit, UnitAction, UnitActionAssignment
    from MicroRTS_NB import UnitType, UnitTypeTable, AStarPathFinding
    from util.buildAction import BuilderAction
    from util.fuctionsUtil import FuctionsUtil

## GameState --- what you receive; read-only

    gs.getPhysicalGameState() -> PhysicalGameState   the map and its units
    gs.getPlayer(player)      -> Player             player is 0 or 1
    gs.getUTT()               -> UnitTypeTable
    gs.getTime()              -> int                current cycle
    gs.free(x, y)             -> bool               no unit there AND not
                                                    reserved by an action in
                                                    flight. BOUNDS-CHECK FIRST
                                                    (see 02_engine.txt).
    gs.getActionAssignment(unit) -> UnitActionAssignment or None
                                                    None means the unit is IDLE
    gs.isUnitActionAllowed(unit, action) -> bool     no conflict with actions
                                                    already in flight
    gs.getResourceUsage()     -> ResourceUsage       resources committed this
                                                    cycle
    gs.getNextChangeTime()    -> int
    gs.gameover()             -> bool
    gs.winner()               -> int                 0 / 1, or -1 if undecided

Never call the mutating members (cycle, issueSafe, updateScream) --- see
02_engine.txt rule 2.

## PhysicalGameState --- the map

    pgs.getWidth()  -> int          READ THESE and guard every coordinate
    pgs.getHeight() -> int
    # Reading the state is order-free; ACTING on it is not --- a unit takes one
    # intention per cycle and the last rule to claim it wins. See 08_structure.
    pgs.getUnits(p) -> dict[int, Unit]
```

```
                                  p = 0 or 1 for a player's units,
                                  p = -1 for the neutral Resource units.
                                  Iterate .values(); the key is the unit id.
    pgs.getUnitAt(x, y) -> Unit or None
    pgs.getUnit(unit_id, player) -> Unit or None
                                  None means that unit is dead --- the way to
                                  check whether a remembered target still
                                  exists.
    pgs.getTerrain(x, y) -> int     0 = open, 1 = wall
    PhysicalGameState.getTERRAIN_NONE() -> 0     (static)
    PhysicalGameState.getTERRAIN_WALL() -> 1     (static)

## Unit

    u.getX(), u.getY()              -> int
    u.getID()                       -> int
    u.getPlayer()                   -> int    owner; -1 for a Resource unit
    u.getType()                     -> UnitType
    u.getHitPoints()                -> int
    u.getMaxHitPoints()             -> int
    u.getResources()                -> int    carried by a worker, or left in a
                                              Resource unit
    u.getAttackRange()              -> int
    u.getMinDamage(), u.getMaxDamage() -> int
    u.getMoveTime(), u.getAttackTime(), u.getHarvestTime(), u.getHarvestAmount() -> int
    u.getCost()                     -> int
    u.toString()                    -> str
    u.getUnitActions(gs)            -> list[UnitAction]

u.getUnitActions(gs) is the safest primitive in the whole API: it returns every
action that is legal for that unit right now, already filtered for map bounds,
occupancy, adjacency and affordability, including a wait action. Choosing from
that list cannot produce an illegal action. For a worker next to a resource it
returns things like: harvest(left), produce(up, Barracks), move(right), wait.

## UnitType

    t.getName()        -> str    "Base","Barracks","Worker","Light","Heavy",
                                 "Ranged","Resource"
    t.getCost()        -> int
    t.getProduceTime() -> int
    t.getMoveTime()    -> int
    t.getAttackRange() -> int
    t.getCanMove(), t.getCanAttack(), t.getcanHarvest() -> bool
    t.getIsStockpile() -> bool   True for Base (resources can be returned to it)
    t.getisResource()  -> bool   True for Resource
    t.equals(other)    -> bool   compare types with this, not with ==

Note the irregular capitalisation of getcanHarvest and getisResource; it is real.

## Player

    p.getResources() -> int   the RAW pool, not counting what actions already in
                              flight will spend. For budgeting use
                              FuctionsUtil.availableResources(gs, player).
    p.getID()        -> int

## UnitTypeTable

    utt.getUnitTypeString(name) -> UnitType     e.g. gs.getUTT()
                                                .getUnitTypeString("Light")
    utt.getUnitTypes()          -> list[UnitType]

## UnitAction --- an action for one unit

Type and direction constants are STATIC methods, not attributes:

    UnitAction.getTYPE_NONE()            0   wait
    UnitAction.getTYPE_MOVE()            1
    UnitAction.getTYPE_HARVEST()         2
    UnitAction.getTYPE_RETURN()          3   give carried resources to a Base
    UnitAction.getTYPE_PRODUCE()         4
    UnitAction.getTYPE_ATTACK_LOCATION() 5

    UnitAction.getDIRECTION_UP()      0      y - 1
    UnitAction.getDIRECTION_RIGHT()   1      x + 1
    UnitAction.getDIRECTION_DOWN()    2      y + 1
    UnitAction.getDIRECTION_LEFT()    3      x - 1
    UnitAction.getDIRECTION_NONE()   -1

    a.getType(), a.getDirection()          -> int
```

```
    a.getLocationX(), a.getLocationY()     -> int   attack target coordinates
    a.getUnitType()                        -> UnitType   for a produce action
    a.getActionName()                      -> str   "wait","move","harvest",
                                                    "return","produce",
                                                    "attack_location"
    a.ETA(unit)                            -> int   cycles to completion
    a.resourceUsage(unit, pgs)             -> ResourceUsage
    a.toString()                           -> str

## BuilderAction --- build UnitActions without the constants

All static. This is the intended way to create actions:

    BuilderAction.makeMove(direction)
    BuilderAction.makeHarvest(direction)      the adjacent cell in that
                                              direction must hold a Resource
    BuilderAction.makeReturn(direction)       the adjacent cell must hold your
                                              Base
    BuilderAction.makeAttack(x, y)            ABSOLUTE coordinates, within range
    BuilderAction.makeProduce(direction, unit_type)
                                              the adjacent cell must be free and
                                              the producer/product pair legal

(The method is makeHarvest. Older copies of this documentation spelled it
makeHavest, which does not exist.)

## PlayerAction --- what you return

    pa = PlayerAction()
    pa.addUnitAction(unit, action)  -> None
    pa.getActions()                 -> dict[int, (Unit, UnitAction)]
    pa.getResourceUsage()           -> ResourceUsage
    pa.setResourceUsage(ru)         -> None
    pa.consistentWith(ru, gs)       -> bool   would adding an action with that
                                              resource usage still be legal
    pa.contain(unit_id)             -> bool

## ResourceUsage

    ru.getResourcesUsed(player) -> int
    ru.consistentWith(other, gs) -> bool
    ru.merge(other)              -> None
    ru.getPositionsUsed()        -> list[int]   cells reserved, encoded as
                                                x + y * width

## FuctionsUtil

    FuctionsUtil.availableResources(gs, player) -> int

The player's pool minus what actions already in flight committed. This is the
number to budget your productions against.

The GameState has no notion of "you": it is the same object for both sides.
Which player you are is ONLY the `player` argument passed to getActions, and the
opponent is `1 - player`. Never look for it on the state.

# Members that do NOT exist

These are real mistakes made by models writing agents for this engine. Each one
raises AttributeError or TypeError on the first cycle and forfeits the match.

    u.getAttackDamage()   does not exist -> u.getMinDamage(), u.getMaxDamage()
    u.getDamage()          does not exist -> same
    u.getHP()              does not exist -> u.getHitPoints()
    u.getMaxHP()           does not exist -> u.getMaxHitPoints()
    u.getPos()             does not exist -> u.getX(), u.getY()
    u.getOwner()           does not exist -> u.getPlayer()
    u.isIdle()             does not exist -> gs.getActionAssignment(u) is None
    t.canHarvest()         does not exist -> t.getcanHarvest()  (lowercase c)
    t.isResource()         does not exist -> t.getisResource()  (lowercase i)
    t.getHitPoints()       does not exist on UnitType -> read it from a Unit
    pgs.getUnitsAt(x, y)   does not exist -> pgs.getUnitAt(x, y)  (singular)
    pgs.getAllUnits()      does not exist -> getUnits(0), getUnits(1), getUnits(-1)
    gs.isFree(x, y)        does not exist -> gs.free(x, y)
    gs.getPlayerResources(p) does not exist -> gs.getPlayer(p).getResources()
    gs.current_player      does not exist -> the `player` argument you receive
    gs.getActivePlayer()   does not exist -> same
    gs.getEnemyPlayer()    does not exist -> 1 - player
    BuilderAction.makeHavest does not exist -> makeHarvest
    BuilderAction.makeBuild  does not exist -> makeProduce(direction, unit_type)
    BuilderAction.makeAttack(unit)  wrong -> makeAttack(x, y), absolute coordinates
```

```
    BuilderAction.makeMove(x, y)    wrong -> makeMove(direction), a direction
                                             constant, one cell

Also: DO NOT CONSTRUCT ENGINE OBJECTS. Unit, GameState, PhysicalGameState,
UnitType and Player come from the engine and their constructors take positional
C++ arguments that will not match what you guess:

    # WRONG --- a model wrote this to reuse a distance helper that calls getX():
    fake = Unit(next_x, next_y, 0, None, 0, 0, 0, 0, 0, 0, 0, 0)

Use plain Python for geometry --- integers, tuples, your own small helpers that
take (x, y) instead of a Unit. Creating a PlayerAction() is the one exception:
that is the object you must build and return.


# The abstraction layer --- the recommended way to write an agent

Writing an agent directly on UnitActions means writing your own pathfinding,
your own bounds checks and your own resource budgeting, every time. The project
ships a layer that does all three. PREFER IT unless you have a reason not to.

    from ai.abstraction.AbstractionLayerAI import AbstractionLayerAI

    class GeneratedAgent(AbstractionLayerAI):
        def __init__(self, pgs, utt):
            super().__init__(pgs)      # builds the A* pathfinder for this map
            self.utt = utt

        def getActions(self, gs, player):
            self.resert()              # forget last cycle's intentions
            ...                        # issue high-level commands, best first
            return self.translateActions(player, gs, PlayerAction())

The idea: you express an INTENTION per unit ("harvest that resource", "attack
that unit", "build a barracks near here"), which can take dozens of cycles, and
translateActions turns the intentions into one legal PlayerAction for this cycle.
Pathfinding, adjacency, direction constants and resource consistency are handled
for you.

## ONE INTENTION PER UNIT PER CYCLE --- the mistake that quietly disables an agent

Each command below records an intention for one unit, and a second command for the
same unit in the same cycle SILENTLY REPLACES the first. So a rule that runs later
cancels the work of a rule that ran earlier:

        self.build(worker, barracks, x, y)     # rule 2 wants a Barracks
        ...
        self.harvest(worker, resource, base)   # rule 5 overwrites it: no Barracks

That agent harvests forever and never builds anything, every cycle, from the first
to the last. It looks like a deliberate economic strategy and it is a bug.

`gs.getActionAssignment(unit)` DOES NOT PROTECT YOU. It reports what the ENGINE is
executing, and an intention you registered this cycle has not reached the engine
yet --- translateActions does that at the end. Worse, if you called `self.resert()`
at the top of getActions, every unit is engine-idle at that moment, so the check
passes for all of them.

The predicate you want is the layer's own:

        if self.getAbstractAction(unit) is not None:
            continue        # already commanded THIS cycle --- leave it alone

Put that check in EVERY rule that can touch a unit an earlier rule may already
have commanded (harvest, fallback attacks, "keep everyone busy" rules). Measured
consequence of omitting it: in one search run, 98 of 99 working programs built no
Barracks at all and never fielded a single combat unit, all from this one missing
line.

## Commands

None of the commands returns an action --- they only register the intention.

# Each call below REGISTERS an intention. Two of them for the same unit in the
# same cycle is not two behaviours: the second silently replaces the first, so
# the SEQUENCE your rules run in is what decides which one survives. Reordering
# the same rules produces a different agent --- that is a knob worth turning.

    self.move(u, x, y)
        Walk to (x, y) by A*. Done when the unit stands there.
```

```
    self.train(u, unit_type)
    self.train(u, unit_type, preference)
        Produce a unit of unit_type from the structure u, in the best free
        adjacent cell: nearest to a resource when producing a harvester, nearest
        to the enemy otherwise. `preference` (a UnitAction direction constant)
        forces one side. Done after a single attempt, so re-issue it every cycle
        if you want continuous production.

    self.build(u, unit_type, x, y)
        Send worker u to (x, y) and build unit_type there. Done when any unit
        occupies (x, y).

    self.harvest(u, resource_unit, base_unit)
        The whole harvesting loop: with empty hands, walk to the resource and
        harvest it; carrying resources, walk to the base and return them. Done
        when that resource unit is exhausted (then pick a new one).

    self.attack(u, target_unit)
        Takes a UNIT, not a coordinate. Within range it attacks the target's
        current cell; otherwise it paths until in range. Done when the target
        dies. Note the target can step away while the attack resolves, so a
        moving target sometimes takes no damage.

    self.idle(u)
        The unit does nothing. CAVEAT: this class's Idle action is meant to
        auto-attack an enemy that is already in range, but the method
        implementing it is misspelled in the project source (`execulte`), so it
        never runs. Do not use idle() as a defensive stance --- issue a real
        attack instead.

## Turning intentions into a PlayerAction

    pa = self.translateActions(player, gs, PlayerAction())

Call it exactly once, at the end of getActions, and return the result. It:
- drops intentions whose unit is dead, and intentions already completed;
- skips units that are still busy (so you never double-command a unit);
- asks each remaining intention for the concrete UnitAction of this cycle;
- accumulates resource usage and adds an action ONLY if the PlayerAction is
  still consistent with it.

That last point is why you do not have to budget resources by hand here. It also
means **the order in which you issue commands is a priority order**: when the
pool runs out, the intentions registered later are the ones dropped. Issue what
matters most first.

## Managing intentions between cycles

    self.resert()      Forget every intention. Call it at the top of getActions
                       to re-decide the whole game state from scratch each cycle.
                       Simplest, and what the example agent does.

    self.clear(gs)     Keep the intentions whose unit is alive and idle, drop the
                       rest. Use this to let long commands (a walk across the
                       map, a harvest loop) run to completion without being
                       reconsidered, and only give orders to units that have
                       none:
                           if self.getAbstractAction(u) is None: ...

    self.getAbstractAction(u) -> the pending intention for u, or None.

Both patterns are valid. resert() costs a little more computation and lets you
react every cycle; clear(gs) is stickier and cheaper.

## Placing buildings

    pos = self.findBuildingPosition(reserved, desired_x, desired_y, p, gs)

Spirals outwards from (desired_x, desired_y) and returns the first free cell as
an ENCODED position, `x + y * width`, or -1 when the map has none. Decode with
`x = pos % width`, `y = pos // width`. `reserved` is a list of encoded positions
you want to avoid; `p` is gs.getPlayer(player).

    self.buildIfNotAlreadyBuilding(u, utype, desired_x, desired_y,
                                   reserved_positions, p, gs) -> bool

Finds a position near (desired_x, desired_y), issues the build, and appends the
chosen position to reserved_positions --- so passing the same list for several
workers keeps them from building on top of each other. Returns False (and does
nothing) if the unit is already building that same type, which makes it safe to
call every cycle. Guard against a full map: it does not check for -1.
```

````
## Pathfinder, if you want it directly

    from MicroRTS_NB import AStarPathFinding
    pf = AStarPathFinding(pgs.getWidth(), pgs.getHeight())

    pf.findPathToAdjacentPosition(unit, position, gs, ru) -> UnitAction
    pf.findPathToPositionInRange(unit, position, range, gs, ru) -> UnitAction

`position` is encoded as `x + y * width`, `ru` is a ResourceUsage (pass the one
you are accumulating, or a fresh ResourceUsage()). The result may be a
TYPE_NONE action, meaning no step is available --- check
`action.getType() != UnitAction.getTYPE_NONE()` and
`gs.isUnitActionAllowed(unit, action)` before using it.

## What the layer does NOT do for you

- It does not decide anything. Build order, unit mix, target selection, when to
  defend and when to push are entirely yours --- that is where a match is won.
- It does not stop you from starving your economy.
- It does not make idle() defend (see the caveat above).
- It does not remove the bounds rule if you also compute coordinates yourself
  (rule 1 in 02_engine.txt still applies to any gs.free call you write).


# Working methods, one at a time --- not an agent

What follows are individual methods from a real agent that plays. They are given
separately, in no particular order, and there is no `getActions` here: deciding
which of them to keep, what else to write, and above all WHICH ORDER to call them
in is the part left to you, and the order changes what the agent does.

Each is a mechanism, not a recommendation. Nothing below says a rule is worth
having, that one unit type beats another, or that these are all the rules there
are.

Two things every one of them relies on:

```python
    def _commanded(self, unit: Unit) -> bool:
        """True if this unit already has an intention THIS cycle.

        A second intention for the same unit REPLACES the first, silently. So a
        rule that runs later cancels an earlier one unless it checks this.
        gs.getActionAssignment does NOT answer this question --- that reports what
        the ENGINE is executing, and an intention registered this cycle has not
        reached the engine yet.
        """
        return self.getAbstractAction(unit) is not None
```

```python
    def _closest(self, unit: Unit, candidates: list):
        """Nearest of `candidates` to `unit`; None when the list is empty.

        Distances are SQUARED --- never compare one against a plain distance.
        Nearest is one way to choose among many; weakest, most dangerous, or a
        fixed coordinate are others, and this does not claim nearest is better.
        """
        best, best_d = None, -1
        for other in candidates:
            d = (unit.getX() - other.getX()) ** 2 + (unit.getY() - other.getY()) ** 2
            if best is None or d < best_d:
                best, best_d = other, d
        return best
```

---

## A method that produces from every structure it owns

```python
    def _rule_train_army(self, world: dict, unit_type) -> None:
        """Ask each Barracks for one unit of `unit_type`.

        A structure is busy for its whole produceTime and a request issued while
        it is busy is discarded in silence --- issuing costs nothing and proves
        nothing. If you want to know how many units you have, count the board.
        """
        for barracks in world["Barracks"]:
            if not self._commanded(barracks):
````
````

````
                self.train(barracks, unit_type)
```

---

## A method that keeps the economy running

```python
    def _rule_harvest(self, world: dict) -> None:
        """Send every free worker to mine and bank one resource.

        self.harvest takes THREE arguments --- the worker, the resource Unit, and
        the structure to carry it back to. Two arguments is a TypeError and the
        match is lost. One call covers the whole round trip over many cycles, so
        re-issuing it to a worker already doing it is wasted.
        """
        if not world["Base"] or not world["Resource"]:
            return
        for worker in world["Worker"]:
            if self._commanded(worker):
                continue
            self.harvest(worker, self._closest(worker, world["Resource"]),
                         self._closest(worker, world["Base"]))
```

---

## A method that reads the board into lists, once

```python
    def _perceive(self, gs: GameState, player: int) -> dict:
        """Split the state by role. Called once; every rule reads these lists.

        pgs.getUnits(p) returns a dict keyed by unit id, so `.values()` is not
        optional: iterating the dict yields INTEGERS and the first .getType()
        raises AttributeError. Owner -1 is neutral --- the resource patches.
        """
        pgs = gs.getPhysicalGameState()
        world = {"Base": [], "Barracks": [], "Worker": [], "Army": [],
                 "Enemies": [], "Resource": []}
        for unit in pgs.getUnits(player).values():
            name = unit.getType().getName()
            if name in ("Base", "Barracks", "Worker"):
                world[name].append(unit)
            else:
                world["Army"].append(unit)
        for unit in pgs.getUnits(1 - player).values():
            world["Enemies"].append(unit)
        for unit in pgs.getUnits(-1).values():
            world["Resource"].append(unit)
        return world
```

---

## A method that sends fighters at something

```python
    def _rule_attack(self, gs: GameState, world: dict) -> None:
        """Give each free fighter an enemy to attack.

        self.attack takes a UNIT as the target, not a coordinate. A unit already
        executing something is skipped: gs.getActionAssignment(unit) is None is
        the engine's own view of "free". Which enemy each one is sent at is not
        decided here.
        """
        if not world["Enemies"]:
            return
        for soldier in world["Army"]:
            if self._commanded(soldier) or gs.getActionAssignment(soldier) is not None:
                continue
            target = self._closest(soldier, world["Enemies"])
            if target is not None:
                self.attack(soldier, target)
```

---

## A method that puts up a structure

```python
````

````
    def _rule_build(self, gs: GameState, player: int, world: dict,
                    structure_type) -> None:
        """Have one free worker start a structure near where it stands.

        buildIfNotAlreadyBuilding finds a free cell near (x, y) itself, so it
        cannot place out of bounds, and does nothing if that worker is already
        building. The sixth argument is the Player OBJECT from gs.getPlayer(player),
        not the integer; the fifth is a list of cells to keep clear, and may be
        empty. Guard the call, or a worker starts one every cycle.
        """
        me = gs.getPlayer(player)
        if me.getResources() < structure_type.getCost():
            return
        for worker in world["Worker"]:
            if self._commanded(worker):
                continue
            self.buildIfNotAlreadyBuilding(worker, structure_type,
                                           worker.getX(), worker.getY(),
                                           [], me, gs)
            return
```

---

## A method that replaces a lost worker

```python
    def _rule_train_worker(self, world: dict, worker_type) -> None:
        """Ask a Base for a worker. How many to keep is not decided here."""
        for base in world["Base"]:
            if not self._commanded(base):
                self.train(base, worker_type)
```

---

# Assembling them is the work, and the order is part of it

None of the above runs on its own. They need a class that subclasses
AbstractionLayerAI, a constructor that keeps `utt`, and a `getActions` that calls
`self.resert()` first and `return self.translateActions(player, gs, PlayerAction())`
last, with whichever rules you want in between.

That "in between" is the decision. A unit takes ONE intention per cycle and the
last rule to claim it wins, so two of these methods calling the same unit are not
two behaviours --- the later one is the behaviour. Calling them in a different
sequence produces a different agent, and one of those sequences plays better than
the others.


# Your opponent

You know nothing about your opponent, and you will not be told anything about it.

What is true of every opponent in this project:

- It is a fixed program. It does not learn, it is not adapting to you across
  matches, and it plays the same way every time from the same position.
- It plays by exactly the rules and the API described in these documents. It has
  no hidden information and no extra powers.
- It could be anything from a bot that never moves to one stronger than anything
  you will write.

You therefore cannot aim at a specific weakness, and nothing else in these
documents hints at one either. The program you are given to modify is your own
lineage, not a portrait of the adversary --- do not read its choices as a claim
about what the opponent does, and do not tune against a strategy nobody described
to you.

What to do instead:

1. **Write a sound general agent.** Keep the economy running, keep producing,
   defend what you cannot afford to lose, do not throw units away, and never
   leave a useful unit idle. Strategies that are good against everything beat
   strategies that are sharp against one guess.
2. **Read your opponent off the STATE, at run time, not from these documents.**
   Everything about it is observable while the match runs. An agent that
   branches on what it observes adapts to an opponent nobody described to it.
3. **Learn from the measured feedback between revisions.** Each version of your
   agent is scored on real matches, and the result --- win, draw or loss per seat,
   plus how much material was left on the board --- is reported back to you. That
````
````

```
   number is the only information about your opponent that exists. A change that
   moved it was right about something; a change that did not was wrong about
   something.

Losing the early attempts is expected. Surviving longer with more material on the
board is real progress and is measured as such, even when the score does not move
yet.


# The structure of an agent, and why order decides what it does

A program for this game is a SEQUENCE OF RULES run once per cycle. Its structure
is not decoration around the strategy --- it IS the strategy, and two programs with
identical rules in a different order are different agents.

## A rule has three parts, and all three are yours to change

    for unit in <SELECTION>:          # which units this rule is about
        if <GUARD>:                   # whether it fires at all
            self.<INTENTION>(unit, ...)   # what it asks for

Changing the selection, the guard or the intention changes the agent. So does
changing nothing about them and moving the rule.

## Order is priority, because a unit takes ONE intention per cycle

A second intention for the same unit in the same cycle silently REPLACES the
first. Two consequences, and both are load-bearing:

- Of two rules that select the same unit, THE LATER ONE WINS. The earlier one is
  not a fallback; it may as well not exist.
- Therefore moving a rule earlier demotes it, and moving it later promotes it ---
  the reverse of how a priority list usually reads.

There are three ways to express priority, and they are not interchangeable:

    # (a) an if/elif chain: one unit, one decision, priority top to bottom
    for worker in workers:
        if <should build>:      self.build(...)
        elif <should harvest>:  self.harvest(...)
        else:                   self.attack(...)

    # (b) separate rules plus a guard, so an earlier claim is respected
    for worker in workers:
        if not self._commanded(worker) and <should build>:
            self.build(...)
    for worker in workers:
        if not self._commanded(worker) and <should harvest>:
            self.harvest(...)

    # (c) separate rules with NO guard --- the last one silently wins
    for worker in workers:
        self.build(...)           # dead: overwritten below, every cycle
    for worker in workers:
        self.harvest(...)

Form (c) is the single most common way a generated agent quietly does less than
its code claims. It reads as two behaviours and executes as one.

## Rules that cannot fire

A rule can be present, correct, and never run. Two ways it happens:

- Its GUARD is never true. A limit compared against a counter you increment
  yourself will stop the rule forever once the counter passes it, whatever the
  board looks like.
- It is nested under a branch that excludes it. A rule placed in the `else` of
  "do I already have one of these?" can only ever create the FIRST one.

Neither shows up as an error, and neither shows up in the score directly --- the
agent simply behaves like a smaller program than it is.

## Counters: count what EXISTS, not what you asked for

A production request the engine cannot execute --- the producer is busy, the cell
is taken, the pool is short --- is discarded in silence. So this is a trap:

    if self.made_soldiers < LIMIT:        # counts REQUESTS
        self.train(barracks, soldier_type)
        self.made_soldiers += 1           # ...including the discarded ones
```

```
The producer is busy for its whole produceTime, so nearly every request during
that window is thrown away and still consumes the limit. Count the board instead:

    soldiers = [u for u in pgs.getUnits(player).values()
                if u.getType().getName() == "Heavy"]
    if len(soldiers) < LIMIT:
        self.train(barracks, soldier_type)

Programs in this project have ranged from three methods and no rules at all to
eighteen methods with a full priority chain. More rules is not better. A rule that
has an effect is better than three that do not.


# Calls that are easy to get wrong, in the form that actually works

Every item below is a mistake that has cost real matches. The signatures are in
03_api.txt and 04_abstraction.txt; what follows is the same information as USAGE,
because the wrong forms are the intuitive ones and reading a signature has not
been enough to displace them. There is no strategy here --- nothing about what to
build, when, or where.

## harvest takes THREE arguments

The intuitive two-argument call is the single most common failure:

    self.harvest(worker, resource)                # WRONG: TypeError, match lost
    self.harvest(worker, resource, base)          # right

The third argument is where the ore is carried BACK to. Both of the last two are
Unit objects, not coordinates, and both have to be found first:

    resource = <some unit from pgs.getUnits(-1).values()>
    base     = <some unit of yours whose type name is "Base">
    self.harvest(worker, resource, base)

## getUnits always takes an owner, and always needs .values()

    pgs.getUnits()                    # WRONG: TypeError, no such overload
    pgs.getUnits(player)              # a dict keyed by unit id
    for u in pgs.getUnits(player).values():        # right: iterate the VALUES

Iterating the dict itself yields the integer keys, and the first `u.getType()`
raises `AttributeError: 'int' object has no attribute 'getType'`. The three owners
are `player`, `1 - player`, and `-1` for the neutral resource patches.

## Unit types are objects, not strings

    self.train(barracks, "Heavy")                       # WRONG
    self.train(barracks, self.utt.getUnitTypeString("Heavy"))    # right

Look the type up once in `__init__` and keep it, or look it up where you use it,
but pass the object. The same applies to `self.build(...)`.

## A UnitActionAssignment is read with a getter

    gs.getActionAssignment(u).action                # WRONG: no such attribute
    gs.getActionAssignment(u).getUnitAction()       # right

Most of the time you do not need the action at all --- you need to know whether the
unit is free, and that is `gs.getActionAssignment(u) is None`.

## buildIfNotAlreadyBuilding wants the Player OBJECT

    self.buildIfNotAlreadyBuilding(u, utype, x, y, [], player, gs)         # WRONG
    self.buildIfNotAlreadyBuilding(u, utype, x, y, [], gs.getPlayer(player), gs)

`player` is an int; the call needs what `gs.getPlayer(player)` returns. The fifth
argument is a list of positions to keep clear and may be empty. The helper finds a
free cell near (x, y) itself, so it will not place a structure out of bounds.

## Members that have been invented and do not exist

    u.isIdle()                    -> gs.getActionAssignment(u) is None
    utype.getcanAttack()          -> utype.getCanAttack()   (capital C)
    utype.getCanHarvest()         -> does not exist
    utt.getUnitType(name)         -> utt.getUnitTypeString(name)
    gs.getAvailableResources(p)   -> FuctionsUtil.availableResources(gs, p)
    self.getAbstractActions()     -> self.getAbstractAction(unit)   (singular)

## An attribute you set in __init__ must survive every edit you make
```

```
Rewriting `__init__` while other methods still read one of its attributes gives
'AttributeError: 'GeneratedAgent' object has no attribute '<name>'' on the first
cycle, and every match is lost. This happens most often when the change you are
making is about numbers or positions and you rewrite the constructor to hold new
ones. Before answering, check that every `self.<name>` you read is still assigned.

If you cannot guarantee that, do not cache it: derive what you need from the state
inside the method that uses it. A value recomputed each cycle is never stale and
never missing.

## The engine discards silently, so a call is not an effect

An action the engine cannot execute --- the producer is busy, the cell is taken, the
pool is short --- is dropped with no exception and no message. Two consequences:

- issuing a request costs nothing, but proves nothing;
- a counter you increment next to the call counts REQUESTS, not units, and will
  stop your own rule long before the board says it should. Count the board.

## Order is not a detail

Every call above registers an intention for one unit, and a unit takes ONE per
cycle: a later call REPLACES an earlier one for the same unit, silently. So the
order your rules run in is part of the program's behaviour, not its formatting.
Two agents with identical rules in a different sequence play differently, and
reordering is one of the cheapest changes you can make.
```

C.2 CANDIDATE-MUTATION PROMPT

The template into which the current candidate program and its evaluation feedback are instantiated at runtime.

Candidate-Mutation Prompt

```
Produce a MODIFIED VARIANT of the MicroRTS agent below.

This is a mutation, not a polish. The variant does NOT have to be better than what
you are given --- it has to be DIFFERENT in a specific, deliberate way, and it has to
run. A search evaluates it and keeps it only if it measures better, so a cautious
rewrite that plays identically is worthless: it costs a full evaluation and teaches
the search nothing. Two variants of the same program that produce the same units and
fight the same way are the same program.

Keep the interface: a class named GeneratedAgent whose only public method is
getActions(self, gs, player) -> PlayerAction, with the same constructor signature.

# What to change

The change for THIS variant is named in the feedback below. Make that one, and make
it concretely: something a reader of the two files could point at, and something
that shows up in what the program produces. A variant that plays identically to its
parent costs a full evaluation and teaches the search nothing.

Add or change ONE mechanism, not several. Say in the class docstring what you
changed and from what to what, so a later mutation can tell what has been tried.

# What NOT to change

- Do not rename or restructure for style. Renaming methods and rewording comments
  produces a new file that plays identically --- the worst possible outcome.
- Do not fix what is not broken. If a rule works, leave it and mutate elsewhere.
- Do not violate the hard rules, and do not drop the guards that keep the agent
  working: one intention per unit per cycle, bounds-checked coordinates, idle-only
  assignment. A mutation that breaks the agent scores zero and is discarded.

Feedback (optional): <e.g. "scores 0.00 over 1 match against the opponent (0 wins,
0 draws, 1 losses) similarity 0.63 to the reference agent">

Current code:
```python
<PASTE THE CURRENT STRATEGY HERE>
```

Output format

Reply with ONE Python code block and nothing else --- no prose outside the block.
```

```
Output the COMPLETE modified file: every import, every method, including the parts
you did not touch. Never a diff, never a fragment, never "unchanged from the
previous version". Complete does not mean large --- most of the file should be
byte-identical to what you were given.
```

C.3 Dynamic Mutation Instructions

One of the following mutation instructions is selected for the current candidate and inserted into the feedback shown in Appendix C.2.

Dynamic Mutation Instructions

```
One of the following mutation directions is selected for the current search
step and appended to the candidate-mutation template as the "Feedback"
mutation instruction.

1. MUTATE THE UNIT TYPE. Whatever combat unit this program trains, train a
different one -- Light (2 cost, 4hp, 2 damage, fast), Heavy (3 cost, 8hp,
4 damage, slow) or Ranged (2 cost, 1hp, 1 damage, range 3) -- or mix two, or
switch type as the match progresses. Adjust the counts that this implies, and
nothing else. The variant must train something else.

2. MUTATE THE QUANTITIES. Keep every rule and every ordering exactly as they
are, and change only the numbers the program already contains -- the limits it
compares against, the thresholds that gate its rules, the distances it
measures. Move them decisively: double one, halve another, drop one to zero,
remove a limit entirely. Not by one.

3. MUTATE THE POSITIONS. Where the Barracks is placed relative to the Base and
to the enemy, where units wait or gather before moving out, how far from home
it defends, which resource each worker mines, whether structures screen the
economy. Same units, same counts, different geometry.

4. MUTATE THE ATTACK STRATEGY. Change which target is chosen (closest,
weakest, the biggest threat, structures instead of units, the enemy economy
instead of its army). Leave production alone.

5. MUTATE THE DEFENCE. Change what the program protects and how: pull workers
out of reach when an enemy closes, keep part of the army home, retreat damaged
units instead of trading them, or drop defence entirely and commit everything
forward. Whichever it does now, do the opposite.

6. MUTATE THE ORDER OF THE RULES. Change nothing about what the rules do and
everything about the sequence they run in. Order is not cosmetic here: a unit
takes ONE intention per cycle, so of two rules that select the same unit, the
one that runs later wins and the earlier one may as well not exist. Moving a
rule earlier promotes it; moving it later can silence it completely. Pick the
rules that compete for the same units and reorder them.

7. MUTATE THE RULE SET. Add a rule the program does not have, or delete one it
has, or change the guard that decides when a rule fires -- the condition, not
the constant inside it. A guard that is never true makes a rule invisible: the
code reads as a feature and behaves as nothing. Prefer one decisive change to
three timid ones.

8. MAKE THE BOLDEST STRUCTURAL CHANGE you can that still runs: a different
build order, a different win condition to aim at, a rule the program does not
have at all. This is the exploration slot -- a variant that scores worse but
plays visibly differently is a better outcome here than a cautious rewrite.

9. TRAIN MORE UNITS. Whatever this program produces, make it produce more of
it: raise or delete the limit that stops production, train every cycle
instead of once, produce from every structure that can produce rather than
one, and stop spending on anything that competes with production. If a guard
blocks training when resources are short, let it try anyway -- the engine
drops what cannot be paid for, at no cost to the rest of the turn.

10. GET A BARRACKS BUILT AND KEEP IT. A Barracks is what lets anything but a
Worker exist, and ordering the build is not the same as finishing it: a
worker that is told to harvest in the same cycle abandons the site, and a
worker that dies mid-build leaves nothing. Make the build actually complete --
give it a worker that no other rule can claim, place it where there is room,
order it before the rules that compete for that worker, and retry while none
exists.
```

C.4 Dynamic Evaluation Feedback

The evaluation feedback varies by guidance condition. Every evaluated candidate provides task-level evaluation information such as score and win/draw/loss results. When evaluation fails, the observed implementation error may be included in the subsequent feedback. Conditions using behavioural guidance additionally provide their corresponding behavioural similarity. No production-count profile is supplied in the reported experiments. The runtime template, with placeholders for the values instantiated at each search step, is:

Dynamic Evaluation Feedback

```
The current program scores <SCORE> over <N> match(es) against <OPPONENT>
(<WINS> wins, <DRAWS> draws, <LOSSES> losses).

[Its behaviour is <SIMILARITY> similar (0 to 1) to <REFERENCE_DESCRIPTION>.]
    -- included only for guidance conditions that use behavioural similarity.

[<ERROR_COUNT> of those matches were forfeited by a failure in the code:
<ERROR_MESSAGE>. Fixing that is the highest-value change available.]
    -- included only when evaluation of the candidate failed.

Goal: maximize the score versus that opponent under the same engine -- but
THIS variant's job is the change named below, not caution. <MUTATION
DIRECTION>
```

Together, the fixed system message (Appendix C.1), the instantiated candidate-mutation message (Appendix C.2) with the current candidate code, the current evaluation result, the condition-dependent behavioural similarity when applicable, and the selected mutation instruction (Appendix C.3) form the complete model input for one candidate generation. This is not a single literal string repeated at every iteration: the candidate code, evaluation result, and mutation instruction are instantiated anew at each search step. Production counts are not part of this input in the reported experiments.