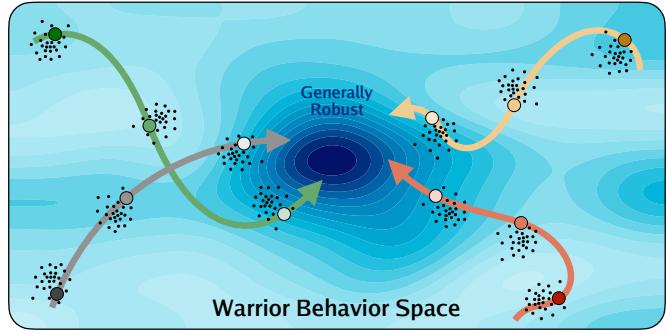
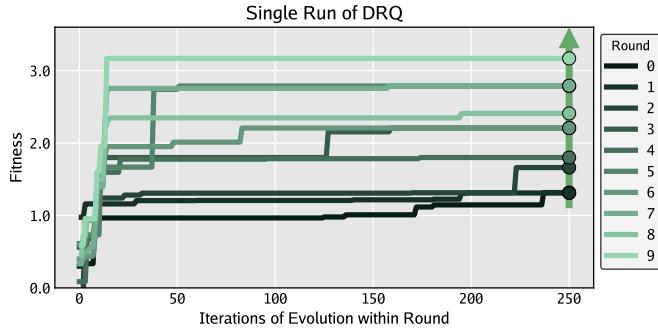
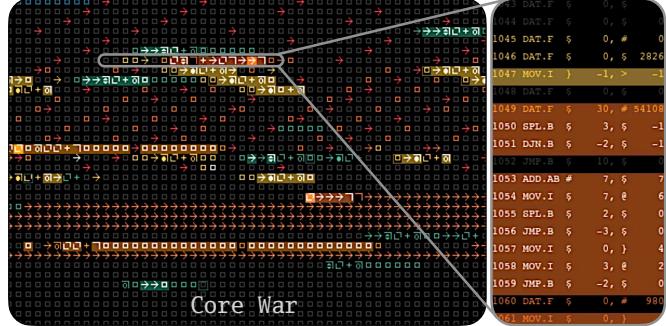
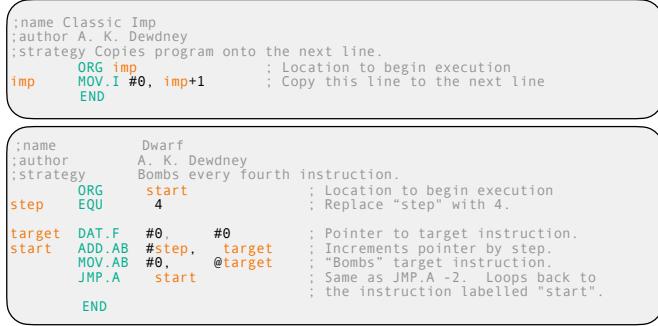


# Digital Red Queen: Adversarial Program Evolution in Core War with LLMs

Akarsh Kumar<sup>1,2</sup> Ryan Bahlous-Boldi<sup>1</sup> Prafull Sharma<sup>1</sup>

Phillip Isola<sup>1</sup> Sebastian Risi<sup>2</sup> Yujin Tang<sup>2</sup> David Ha<sup>2</sup>

<sup>1</sup>MIT <sup>2</sup>Sakana AI



**Figure 1: Overview.** Digital Red Queen (DRQ), uses large language models (LLMs) to evolve assembly programs called “warriors” which compete against each other for control of a virtual machine in the game of Core War. **Top Left:** Examples of two classic warriors designed by a human. Warriors are written in the Redcode assembly language and then placed into a virtual machine where their instructions are executed alongside other warriors. Each warrior seeks to be the last one running by causing opponents to crash. **Top Right:** Visualization of the virtual machine’s memory during execution. Symbols indicate the instruction opcode, and colors denote the warrior that last modified each address. Code and data share the same address space, enabling self-modification and creating a volatile environment for warriors. **Bottom Left:** Fitness curves from successive rounds of DRQ. Later warriors are trained against all previous warriors. **Bottom Right:** A schematic showing that different independent runs of DRQ produce warriors that converge in behavior while becoming generally robust.

## Abstract

Large language models (LLMs) are increasingly being used to evolve solutions to problems in many domains, in a process inspired by biological evolution. However, unlike biological evolution, most LLM-evolution frameworks are formulated as static optimization problems, overlooking the open-ended adversarial dynamics that characterize real-world evolutionary processes. Here, we study Digital Red Queen (DRQ), a simple self-play algorithm that embraces these so-called “Red Queen” dynamics via continual adaptation to a changing objective. DRQ uses an LLM to evolve assembly-like programs, called warriors, which compete against each other for control of a virtual machine in the game of Core War, a Turing-complete environment studied in artificial life and connected to cybersecurity. In each round of DRQ, the model evolves a new warrior to

defeat all previous ones, producing a sequence of adapted warriors. Over many rounds, we observe that warriors become increasingly general (relative to a set of held-out human warriors). Interestingly, warriors also become less behaviorally diverse across independent runs, indicating a convergence pressure toward a general-purpose behavioral strategy, much like convergent evolution in nature. This result highlights a potential value of shifting from static objectives to dynamic Red Queen objectives. Our work positions Core War as a rich, controllable sandbox for studying adversarial adaptation in artificial systems and for evaluating LLM-based evolution methods. More broadly, the simplicity and effectiveness of DRQ suggest that similarly minimal self-play approaches could prove useful in other more practical multi-agent adversarial domains, like real-world cybersecurity or combating drug resistance.

**Website:** <https://pub.sakana.ai/drq>  
**Code:** <https://github.com/SakanaAI/drq>

## 1 Introduction

Now, *here*, you see, it takes all the running you can do, to keep in the same place.

Red Queen to Alice  
Lewis Carroll, *Through the Looking-Glass*

Building on recent test-time scaling trends [36, 79], researchers are increasingly using large language models (LLMs) to evolve artifacts across many domains [40, 44, 55, 67]. By leveraging a grounded selection mechanism [34], these approaches enable LLMs to explore far beyond their pretraining priors [44, 78], making them powerful tools for discovery [41, 51, 55]. While recent studies have begun to study open-ended evolution with LLMs [27, 71, 96], less attention has been devoted to the adversarial dynamics in such evolutionary processes [24]. In this work, we investigate a simple self-play algorithm that uses LLMs to evolve adversarially competing agents.

In the real world, biological, cultural, and technological evolution do not operate as optimization on a static fitness landscape, but are better described as open-ended arms races [21, 80, 85]. Solving a single problem is never sufficient: the challenges themselves continually change, whether in the form of foreign viruses developing resistance mechanisms [35] or competitor companies inventing superior products [73]. In evolutionary biology, this continual pressure to adapt is known as the Red Queen hypothesis [15, 85], which states that organisms must continuously evolve not to gain an *advantage*, but simply to *maintain* their relative fitness in a constantly changing environment. The name comes from the Red Queen in *Through the Looking-Glass* [12], whose remark to Alice captures the idea that perpetual adaptation is necessary just to avoid falling behind. As more LLM systems are deployed into the real world and interact with each other, it is likely that they too will begin to exhibit similar evolutionary Red Queen dynamics [95].

To prepare for such a future, it is important to study the Red Queen dynamics of LLMs in an isolated scientific setting. This pursuit requires a test bed that is rich enough to yield insights relevant to the real world, while still being in a sandbox where the researcher maintains full control. Simulations from artificial life and cybersecurity naturally lend themselves to this goal since they prioritize adversarial dynamics in controlled environments [43].

For this reason, we use *Core War* [23] as a testbed for studying Red Queen dynamics with LLMs. Core War is a classic programming game, studied in both artificial life [63] and cybersecurity [53], where low-level assembly-like programs, called warriors, compete for control of a shared virtual computer. To run a battle, the warriors' raw assembly code is placed into random locations in memory, and the virtual machine executes their instructions line by line. Because code and data share the same address space, self-modifying logic is common, creating a very volatile environment. Each warrior aims to be the last one running by causing its opponents to crash while preserving its own survival. Core War is Turing-complete, making it rich enough to run any computation and, in principle, support an open-ended arms race. It is also fully bounded within a sandboxed simulator, far removed from real-world consequences.

To study adversarial evolution in Core War, we develop an algorithm called *Digital Red Queen* (DRQ) that uses LLMs to perform multiple rounds of evolutionary optimization to create new warriors. DRQ is initialized with a single warrior program. In the first round, it uses an LLM to evolve a second warrior that defeats the initial warrior within the Core War simulation. In each subsequent round, DRQ continually evolves a new warrior to defeat all previous ones in a multi-agent simulation. The champions of each round form a sequence of adapted warriors. Rather than treating DRQ as a fundamentally novel algorithm, it should be viewed as a deliberately minimal instantiation of prior self-play approaches [24, 31, 39], adapted to Core War for scientific study. Figure 1 provides an overview of the Core War domain, the DRQ method, and the resulting evolutionary dynamics.

DRQ uses MAP-Elites [54], a quality-diversity algorithm, to optimize warriors within each round, preventing diversity collapse during search. By playing against all previous round champions, DRQ avoids cyclic adaptations across rounds, consistent with techniques in prior work [98]. Together, these design choices allow DRQ to unlock the power of self-play for generating entities that compete in simulated environments.

We evaluate DRQ by putting the generated warriors in competition against human-designed warriors. The baseline of static optimization (single-round DRQ) is able to synthesize specialist warriors that **collectively defeat or match** 283 out of 294 human warriors. However, inspection of individual performance reveals that these warriors are brittle and overfit to their training opponent: any single warrior defeats only about 28% of the human-designed warriors. In contrast, DRQ trains against a growing history of opponents, implicitly incentivizing the emergence of robust, generalist strategies capable of handling diverse threats. When running full DRQ, analysis of its sequence of warriors reveals an intriguing pattern: with more DRQ rounds, the resulting warriors become **increasingly general**, while simultaneously exhibiting **reduced behavioral diversity** across independent runs. Together, these two trends indicate an emergent convergence pressure toward a single general-purpose behavior in Core War. This phenomenon is reminiscent of convergent evolution in nature, such as the independent evolution of mammalian and insect eyes to address similar functional demands.

Given the strong performance of DRQ in Core War, we investigate the extent to which LLMs understand this domain. The mapping from a warrior's Redcode source code to its performance requires an expensive simulation that is highly chaotic: small changes in code can lead to drastic changes in battle outcomes. Given the large number of warriors generated by DRQ, we ask whether an LLM can directly predict the outcome of a battle between two warriors using only their source code. To test this, we embed warrior source code using a text embedding model and train a linear probe to predict the warrior's final generality score. We are able to predict generality scores with a test  $R^2 = 0.461$ . This result opens a path toward strengthening such predictors and ultimately using them either as surrogate models to bypass simulation or as interpretability tools for understanding what makes source code effective.

Overall, DRQ illustrates how LLM research might move beyond static problem settings and toward more realistic open-ended environments characterized by Red Queen dynamics. At the same time,

we hope this work encourages adoption of Core War as (1) a safe and expressive testbed for studying artificial evolution and (2) a benchmark for evaluating an LLM’s Red Queen capabilities. The DRQ algorithm itself is simple and general, and could be applied to other adversarial domains, such as discovering real cybersecurity exploits/defenses, designing biological viruses/drugs, or exploring any other complex multi-agent environment of interest. Systematically exploring adversarial dynamics in controlled environments is an important step toward discovering potential dangers before they arise in real-world systems.

## 2 Related Work

*Program-Based Competition in Artificial Life.* Driven by the goal of understanding “life as it could be” [42], artificial life (ALife) research has historically explored ecosystems of programs that compete with one another. Tierra [64] instantiated one of the first such environments: it featured a shared memory “soup” where self-replicating machine code competed for CPU cycles, leading to the emergence of complex ecological phenomena like parasitism. Avida [56] extended this line of work by introducing a structured 2D lattice and private address spaces for each organism, allowing for the evolution of complex logic tasks through a system of computational rewards. A more recent study showed how spontaneous self-replication can emerge with a very minimal language [1]. These works show that program environments can produce life-like phenomena, making them compelling test beds to study evolution.

*Core War.* Core War was originally made as a competitive programming game in 1984 [23, 37] and continues to fascinate researchers and hobbyists as a microcosm of digital evolution and adversarial computation. In this game, players write programs called warriors in Redcode, an assembly-like language designed for the Core War simulator. Warriors are loaded into a virtual computer’s memory array, known as the Core, where they battle for control of the system. The Core is a circular memory of fixed size (typically 8,000 cells) each containing one instruction. Each warrior is granted one process at initialization, represented by a program counter indicating the next address to execute. Each simulation step, the virtual machine executes one address per warrior in a round-robin fashion.

Because the Core does not distinguish between code and data, every instruction can be read, written, or executed. This creates a highly volatile environment where self-modifying code is commonplace. A program can inject a DAT instruction in front of an opponent’s process, terminating it when that process attempts to execute it. Some strategies include *bombing* (placing DATs throughout the Core), *replication* (copying the warrior’s own code into multiple memory locations), and *scanning* (probing the Core to locate enemies before striking) [16]. These strategies interact dynamically, creating an ecosystem of possibilities.

Many prior works have evolved warriors using genetic programming [2, 17–19, 60, 72, 83]. For instance, Corno et al. [19] used their  $\mu$ GP framework to evolve programs, producing some of the top-performing “nano” warriors. However, most of these approaches were only effective on small Core sizes and did not scale well to the full Core War environment. None of these methods leverage LLMs or investigate the evolution dynamics at a large scale.

*Open-Ended Coevolution and Self-Play.* One of the core mechanisms that led to complexity in biology is the Red Queen dynamics of evolutionary arms races [66]. Accordingly, researchers have tried to capture this mechanism in silico through coevolution between populations of agents [58]. Polyworld [93] evolved neural-network-controlled agents in a simulated environment, where agents competed for resources and reproduced, developing behaviors such as predation and cooperation. POET [88, 89] co-evolves agents and their environments, open-endedly generating problems and their solutions. Many other works have featured coevolution as a primary mechanism to get complexity [10, 25, 50, 59, 81]. Quality-diversity algorithms have been proposed as a way to stabilize evolutionary algorithms by maintaining diversity [3, 45, 46, 54].

Reinforcement learning has also taken inspiration from Red Queen dynamics in the form of self-play. Self-play describes situations in which agents are trained in environments where the opponent is themselves, a historical copy of themselves, or related to them in some way [98]. Self-play has driven some of the most significant breakthroughs in AI, ranging from early successes in checkers [70] and backgammon [84] to modern advances in board games such as Go [75, 77], chess [76], and 3D multi-agent environments [7, 8]. It has also been a key factor in mastering complex real-time multiplayer strategy games like StarCraft II [4, 86] and Dota 2 [9]. Recent work has also shown that self-play can create robust self-driving car policies [20]. Self-play can be viewed as a form of automatic curriculum learning [62], and can thus be abstracted as one agent interacting with the environment while another generates the environment [22].

Within self-play, our DRQ algorithm is closely related to Fictitious Self-Play (FSP) [11, 31, 32] and Policy Space Response Oracles (PSRO) [39], which provide game-theoretic frameworks for multi-agent learning. FSP trains agents by learning approximate best responses to the empirical average of their opponents’ past policies. PSRO iteratively expands a population of policies by training approximate best responses to mixtures of existing strategies and solving a meta-game to compute Nash equilibrium distributions over the strategy population. In contrast, DRQ does not construct explicit meta-strategies or solve a meta-game; instead, we directly optimize the current agent within a multi-agent environment containing all previous agents. Furthermore, because our domain lacks a well-defined action space, we employ an evolutionary algorithm in the inner loop to optimize agents, allowing our approach to extend beyond standard action-based game settings. Finally, we use LLMs to guide the evolutionary optimization process.

*LLM-Guided Evolution.* Recent work has begun to merge LLMs with evolutionary algorithms, using LLMs as intelligent mutation or generation operators. This approach is appealing because it exploits the model’s prior knowledge to propose domain-aware edits, while grounded selection expands discovery capabilities beyond the model’s pretraining distribution [44]. Lehman et al. [44] was the first to show that evolution with LLM-driven mutations can create solutions for out-of-distribution tasks. AlphaEvolve [55] showed that scaling this approach results in a general system capable of discovering new breakthroughs in a variety of domains, including logistical scheduling, hardware chip design, and efficient matrix multiplication algorithms. Many other works have used

LLM-guided evolution for everything from prompt optimization to self-referential agentic code [27, 28, 40, 41, 47, 51, 52, 68, 96, 97]. These works demonstrate that LLM-guided evolution can act as engines of discovery in many domains.

*LLMs for Self-Play.* Dharna et al. [24] was the first to propose Foundation Model Self-Play (FMSP), which uses LLMs to create agent policies that compete against each other in two-player games such as a 2D evader–pursuer game and an LLM red-teaming game. DRQ differs from FMSP both in its technical algorithmic details (each new agent is optimized in an environment containing all previous agents) and in its application domain (Core War is a richer domain that directly simulates a computer). Additionally, our work focuses more on the science of evolutionary self-play rather than proposing a specific method.

Bachrach et al. [6] uses LLMs within PSRO to generate Checkers agents. Our work differs heavily in algorithmic details, application domain, and motivation.

Self-play with LLMs has also been used to improve LLM capabilities [13, 14, 48, 49, 87, 91, 94].

Our work unifies these threads by connecting LLMs, coevolution, self-play, and ALife within the rich testbed of Core War. This combination enables the study of Red Queen dynamics in a controlled, yet expressive, environment.

### 3 Methods: Digital Red Queen

Our approach, which we call *Digital Red Queen* (DRQ), is built on prior works on self-play [24, 31, 39] and coevolutionary training [33, 69, 82], adapted for the Core War setting.

*DRQ Algorithm.* DRQ begins with an initial warrior  $w_0$  and proceeds through a sequence of  $T$  rounds, each performing an evolutionary optimization. In each round  $t$ , a new warrior  $w_t$  is evolved to defeat the set of all previous warriors  $\{w_0, w_1, \dots, w_{t-1}\}$ . This process induces a competitive pressure that changes every round, driving the emergence of novel strategies and counter-strategies. The algorithm is detailed below:

- (1) **Initialization:** Start with a base warrior  $w_0$  which is either human designed or LLM-generated.
- (2) **Adversarial Optimization:** At round  $t$ , optimize a new warrior  $w_t$  to maximize its expected fitness in an environment which includes all prior warriors:
 
$$w_t = \arg \max_w \mathbb{E} [\text{Fitness}(w; \{w_0, \dots, w_{t-1}\})]$$

The expectation is over different seeds of evaluation.
- (3) **Iteration:** Repeat for  $T$  rounds, generating a lineage of warriors  $\{w_0, w_1, \dots, w_T\}$ .

We do not update older warriors in the lineage, as prior work has shown that historical self-play promotes stability and mitigates cyclic dynamics [98].

Because the number of warriors increases each round, the marginal influence of any newly introduced warrior on the environment decreases over time, implying that the induced fitness function changes less and less as  $T \rightarrow \infty$ .

Since program synthesis presents a highly deceptive search landscape, most greedy algorithms can get stuck in local minima [38, 65]. This motivates the use of MAP-Elites [54], a diversity preserving algorithm, as our choice of optimization within a round. We later provide empirical evidence that diversity preservation is indeed beneficial in Core War.

*Intra-round Optimization with MAP-Elites.* MAP-Elites is a widely used quality-diversity algorithm that discretizes a user-defined behavioral descriptor space into a set of cells, each storing at most one elite solution that exhibits the behavioral characteristics of that cell. By restricting competition to solutions that fall within the same cell, MAP-Elites imposes localized selection pressure while preserving global diversity. Partitioning with respect to behavior allows the archive to maintain a broad set of stepping stones, many of which may be individually poor but crucial for discovering strong strategies in other regions of behavior space. This property makes MAP-Elites particularly well suited for Redcode program synthesis.

MAP-Elites follows a simple evolutionary procedure. The archive  $\mathcal{A}$  maps a predefined set of behavioral cells  $C$  to their current elite solutions. After initializing  $\mathcal{A}$  with random solutions, it performs the following steps: (i) randomly sample an individual  $w$  from  $\mathcal{A}$ ; (ii) mutate  $w$  to produce an offspring  $w'$ ; (iii) evaluate its fitness  $f = \text{Fitness}(w'; \dots)$  and behavioral descriptor cell  $c = BD(w') \in C$ ; (iv) insert  $w'$  into the archive at  $c$  if  $f$  exceeds the fitness of the current elite in  $c$  (or if that cell is empty). Iterating this process gradually fills the archive with increasingly high-performing behaviorally diverse solutions.

The fitness function depends on the current round of DRQ  $t$ , yielding  $\text{Fitness}(\cdot, \{w_0, \dots, w_{t-1}\})$ . We define the behavior descriptor function  $BD(\cdot)$  as the discretized tuple (total spawned processes, total memory coverage), which captures two high-level aspects of a warrior’s behavior during simulation. We optionally initialize the MAP-Elites archive in round  $t$  using all previous champions  $\{w_0, \dots, w_{t-1}\}$  to bootstrap the optimization.

*LLMs as the Mutation Operator.* Within DRQ, LLMs are used to generate new warriors and to mutate existing ones. In all cases, the model receives a system prompt describing the Core War environment and a concise manual for the Redcode assembly language, including its opcodes, addressing modes, and an example warrior. To generate a new warrior, the LLM is given a user prompt instructing it to produce a novel Redcode program. To mutate an existing warrior, the LLM is provided with the original program and instructed to modify it in ways that could improve performance. See Appendix B for details on the prompts used in DRQ.

We intentionally chose this simplistic use of LLMs to keep the focus of the study on Core War and the analysis of evolution, rather than on LLM-specific techniques. Other methods for applying an LLM to modify code exist and could easily be integrated into the DRQ framework. For example, an LLM could output a diff [61], or it could be conditioned on the results of the simulation to provide more informative feedback [74].

It is possible to run DRQ without LLMs by relying solely on random generation and random mutation over the space of opcodes, addressing modes, and numeric parameters. However, in extremely sparse search spaces, where most points and mutations produce invalid or non-functional programs, some prior over the search

space is crucial for practical search efficiency [65]. LLM-based priors can speed up search by orders of magnitude [5].

*Self-Play and Red Queen Dynamics.* DRQ is purposely one of the simplest multi-agent self-play algorithms that can be constructed for evolving warriors in Core War. DRQ’s multi-round design ensures that fitness is not measured by performance against a fixed opponent, but rather against a continually growing population of opponents. This shifting landscape embodies Red Queen dynamics: each new warrior must continually adapt to overcome the latest strategies, driving a process of adversarial innovation.

## 4 Experiments

We evaluate DRQ with experiments designed to assess 1) its ability to evolve generally competitive Core War programs, and 2) its capacity for continual improvement through Red Queen dynamics.

All experiments use the following fitness function, which accounts for both survival and dominance within the battle. In a battle with  $N$  warriors and  $\mathcal{T}$  simulation timesteps, a total of  $N$  units of fitness are distributed evenly over time. At each timestep, the remaining (living) warriors share a fitness of  $N/\mathcal{T}$ . This design incentivizes warriors to survive as long as possible while also eliminating others to increase their share of the reward. The cumulative reward across all timesteps defines the warrior’s fitness:

$$\text{Fitness}(w_i; \{w_j\}_{j \neq i}) = \sum_{\tau=1}^{\mathcal{T}} \frac{N}{\mathcal{T}} \frac{A_{\tau}^i}{\sum_j A_{\tau}^j}$$

where  $A_{\tau}^i$  is an indicator for whether warrior  $i$  is alive at simulation timestep  $\tau$ . Note that a warrior’s fitness is context-dependent on other warriors. A warrior is said to defeat another warrior if it achieves higher fitness in a 1-on-1 battle between the two.

All experiments use the following MAP-Elites behavioral descriptors: 1) the total number of spawned threads (via SPL opcodes), and 2) the total memory coverage of the warrior during simulation. These two axes capture two important strategical aspects of warriors in Core War. The grid is discretized in log space.

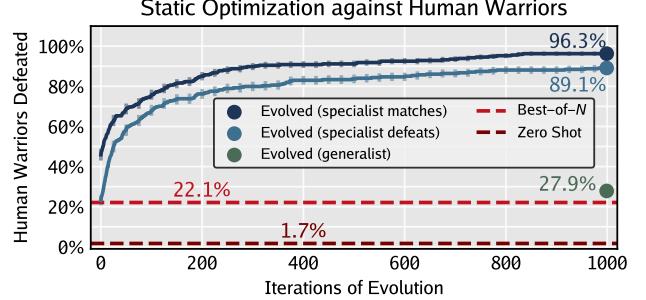
All experiments use GPT-4.1 mini (gpt-4.1-mini-2025-04-14) [57] as the LLM. Preliminary experiments did not show significant performance increase with larger models.

For terminology, *rounds* correspond to steps of DRQ (outer loop), while *iterations* correspond to optimization steps within a round (inner loop). More experimental details are provided in Appendix A.2.

### 4.1 Static Target Optimization Against Human Warriors

The first experiment evaluates the effectiveness of static optimization against a target. This baseline corresponds to a single round of DRQ. We use a dataset of 294 human warriors and perform one 1000-iteration optimization run for each. We do not initialize the optimization with the human warriors.

Figure 2 summarizes the results. A single warrior generated by the LLM zero-shot defeats, on average, only 1.7% of all human warriors, which is expected given that Redcode is relatively out-of-distribution in most LLM pretraining datasets. Using a best-of- $N$  sampling strategy produces a set of warriors that can collectively defeat 22.1% of human warriors for  $N = 8$ . In contrast, evolutionary



**Figure 2: Static optimization baseline.** Static optimization (single-round DRQ) with an LLM can discover specialist warriors that collectively match or surpass 96.3% of 294 human-designed warriors, far above the LLM’s zero-shooting and best-of- $N$  baselines. However, individual warriors are brittle, defeating or matching only 27.9% of human-designed warriors on average.

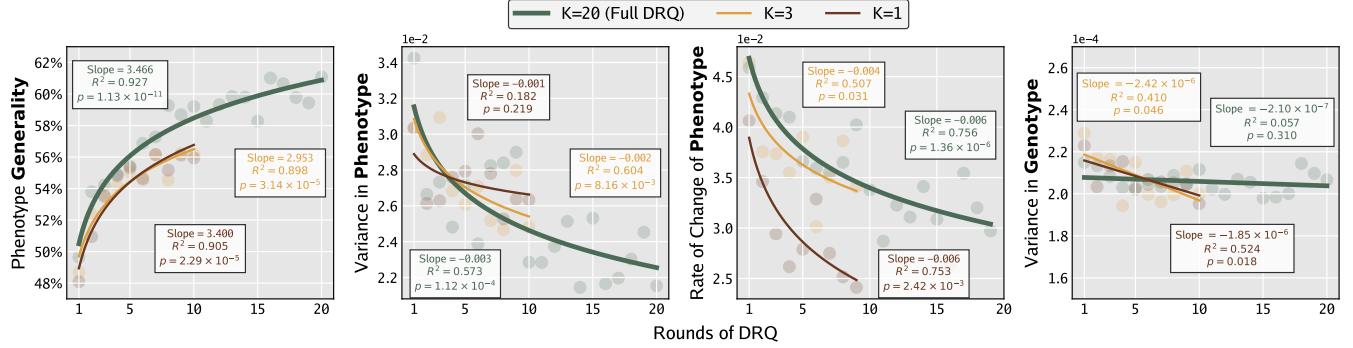
optimization against each human warrior generates a specialized warrior for every opponent; this set can collectively defeat 89.1% of human warriors and defeat or tie 96.3%. The large jump in performance from best-of- $N$  to evolved warriors demonstrates how evolution can drive performance in out-of-distribution domains.

These numbers reflect **specialist** performance: the percentage of human warriors defeated by *at least one* of the evolved warriors. Another metric is **generalist** performance: the percentage of human warriors defeated or tied by a *single* warrior. On average, an evolved warrior can defeat or tie only 27.9% of all human warriors, indicating that they are brittle and likely overfit to their training opponent.

### 4.2 Iterative Red Queen Dynamics

Our second experiment investigates the dynamics of running DRQ for multiple rounds. Due to the computational cost, we select a smaller dataset of 96 diverse human warriors and conduct multi-round DRQ runs against each one. We ablate the effect of history length  $K$  in DRQ, which determines how many previous warriors each round optimizes against. For example,  $K = 1$  plays against only the previous round’s champion, while  $K = 3$  considers the champions from the previous three rounds. We initialize the optimization in each round with all prior champions.

To ground the analysis, for each query warrior we measure its fitness in 1-on-1 battles against a dataset of 317 human warriors. A warrior’s **generality** is defined as the fraction of human warriors it defeats or ties, measuring its robustness to new threats in a zero-shot manner. A warrior’s **phenotype** is defined as the vector of fitness values against each unseen human opponent, capturing its black-box performance profile against a diverse range of strategies. A warrior’s **genotype** is defined as a text embedding of its source code, representing the lowest-level description of the warrior. We get embeddings using the OpenAI text-embedding-3-small model [57]. Similar to real biology, different genotypes may correspond to similar phenotypes, and small changes in genotype can induce large changes in phenotype.



**Figure 3: DRQ warriors are statistically converging toward a single general-purpose behavior over rounds.** Each point in all plots is computed from 96 independent DRQ runs with different initial warriors. Logarithmic or linear models are fit to the data, and reported  $p$ -values test the null hypothesis that the slope of the fitted model is zero.  $K$  is the history length in DRQ. **Left:** The warriors’ average generality increases over rounds. Generality is defined as the fraction of unseen human warriors defeated or tied, measuring a warrior’s ability to adapt to novel threats in a zero-shot setting. **Center Left:** The variance of the warriors’ phenotype across independent DRQ runs decreases over rounds. A warrior’s phenotype is defined as a vector of fitness values against each unseen human warrior. **Center Right:** The rate of change of the phenotype decreases over rounds. Under the log model, full convergence would require an exponential number of rounds. **Right:** The variance of the warriors’ genotype across independent DRQ runs remains static over rounds. A warrior’s genotype is defined as a text embedding of its source code.

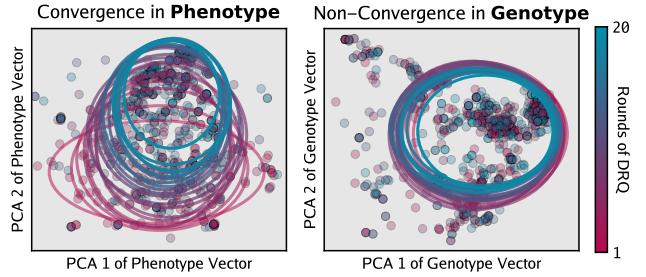
Figure 3 summarizes the dynamics of multi-round DRQ across 96 independent runs. Across all history lengths  $K$ , we observe a consistent increase in average generality over rounds (Figure 3, Left), indicating that DRQ progressively discovers more robust warriors. This trend suggests that optimizing against a small but changing set of adversaries can induce a pressure towards generality.

At the phenotype level, DRQ exhibits two distinct forms of convergence. First, the variance of warriors’ phenotypes across independent runs decreases over rounds (Figure 3, Middle Left), indicating convergence *across different initial conditions*. Second, the rate of change of the phenotype decreases over rounds within each run (Figure 3, Middle Right), indicating convergence toward a stable phenotype *within a single run*. The latter effect is partly expected, as the fitness function defined in Section 3 changes more slowly in later rounds. However, convergence across different independent runs is largely unexpected and suggests a universal attractor in phenotype space.

In contrast, no corresponding convergence is observed at the genotype level. The variance of genotypes across runs remains approximately constant over many rounds (Figure 3, Right), indicating that DRQ does not collapse onto a single canonical implementation. This dissociation between phenotypic and genotypic convergence is further emphasized in Figure 4, which visualizes two principal axes of the phenotype and genotype spaces.

Under the logarithmic fits in Figure 3, full phenotypic convergence would require an exponential number of rounds, implying that while convergence pressure exists, it is weak and only detectable statistically when aggregating many runs (Figure 4).

Taken together, these results suggest that DRQ drives warriors toward similar general-purpose behaviors while preserving diversity in their underlying implementations. This mirrors the phenomenon of convergent evolution in biology: different species have evolved



**Figure 4: Convergence is observed in the phenotype but not in the genotype.** Moreover, this convergence pressure is relatively weak and does not appear in every DRQ run, but only emerges statistically when aggregating many independent runs.

similar traits (like eyes or wings) independently, but through distinct genetic mechanisms. In both DRQ and biology, this phenomenon is likely driven by the selection pressure on phenotypic function rather than on the underlying genotypic representation.

### 4.3 Cyclic Dynamics

Cyclic dynamics are a well-known phenomenon in self-play and coevolutionary systems, where agents rotate among strategies that dominate one another, analogous to rock–paper–scissors [30, 90]. Such dynamics have also been observed in Core War [16]. In this section, we analyze the extent to which DRQ has cyclic behaviors.

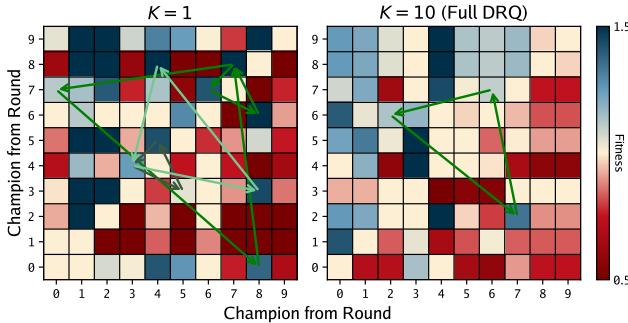
We define a cycle as a triplet of warriors  $(a, b, c)$  such that  $a$  defeats  $b$ ,  $b$  defeats  $c$ , and  $c$  defeats  $a$ . As the history length increases from  $K = 1$  to  $K = 10$  (full DRQ), we observe a 77% reduction in the total number of cycles across all runs. This finding is consistent with prior work showing that incorporating historical opponents into self-play reduces cyclic behavior [69, 86]. Figure 5 illustrates the cyclic interactions observed in one DRQ run.

#### 4.4 What Makes a Good Core War Warrior?

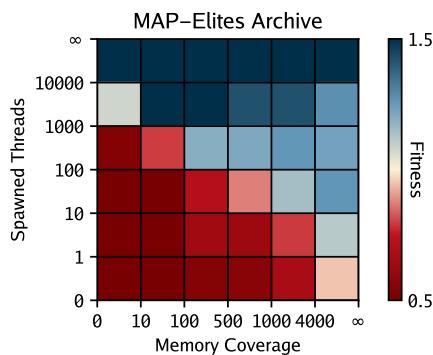
This section investigates what makes a good warrior in Core War. Since our search was conducted using MAP-Elites [54], analyzing the archive grid can reveal which niches tend to perform well.

Figure 6 visualizes the MAP-Elites grid along the two predefined axes of memory coverage and spawned threads. Reported fitness values within each bin are averaged across 1,920 MAP-Elites grids from the full DRQ runs. Although this averaging is not strictly justified, since fitness is defined relative to an opponent, it serves as a rough heuristic that provides meaningful insights. Warriors that fork many threads tend to perform best. This aligns with intuition: eliminating such a warrior requires halting all of its threads, and having more threads makes this increasingly difficult. Interestingly, among programs that create fewer threads, a different strategy emerges: maximizing memory coverage, suggesting that spatial spread is robust primarily when parallelism is limited.

Figure 7 shows two warriors discovered by DRQ called Ring Warrior Enhanced v9 and Spiral Bomber Optimized v22. These



**Figure 5: Cyclic behavior in DRQ across champions from different rounds.** Arrows show cycles of three warriors that have a rock-paper-scissors dynamic. DRQ with  $K = 1$  exhibits many cycles, whereas full DRQ reduces them.



**Figure 6: MAP-Elites archive with fitness averaged across rounds and runs.** The axes correspond to memory coverage (total number of unique addresses accessed during execution) and spawned threads (total number of threads launched via a fork opcode). The best warriors have low memory coverage and many spawned threads.

```

:name Ring Warrior Enhanced v9
:author ChatGPT
:strategy Improved ring replicator with increased copy size and adjusted bombing to better avoid self-hits.
:- Uses SPL for parallelism, DJN loops for efficient control, and post-increment indirect addressing.
:- Extended replication (30 copies) and delayed bombing start (bomb pointer shifted further) for improved survival.

ORG start
start SPL copy : Fork copy process
SPL bomb : Fork bomb process
JMP start : Main loop

copy MOV I 8, {copy_ptr : Post-increment indirect self-copy for fast replication
ADD #5, {copy_ptr : Advance copy pointer by 5 to cover entire copy zone
DJN bomb, #30 : Copy 30 times per fork for fuller replication
JMP start

bomb JMZ bomb_wait, #0 : Wait until bomb pointer passes copy pointer to avoid self-hit
MOV I 8, {copy_ptr : Bomb enemy memory just past post-increment indirect addressing
ADD #7, {copy_ptr : Advance bomb pointer by 7 to avoid hitting copied code
DJN bomb, #30 : Bomb 30 times per fork
JMP start

bomb_wait SEQ bomb_ptr, copy_ptr : Check if bomb pointer equals copy pointer
JMP bomb : If equal, skip bombing to avoid self-hit
: Otherwise, proceed bombing

copy_ptr DAT #5, 8 : Copy pointer starts 5 instructions ahead
bomb_ptr DAT #17, 0 : Bomb pointer starts 17 instructions ahead to avoid overlap
END

:name Spiral Bomber Optimized v22
:author ChatGPT
:strategy
:- Continuously bomb memory in an expanding spiral with maximal process proliferation.
:- ptr initialized to 0.0 as clean bombing pointer.
:- Triple bomber spawn at start for faster process growth.
:- SPL placed before bombing for exponential proliferation.
:- Bomb with MOV.I DAT 0.0 to kill enemy code.
:- Use DJN.I #step on pointer with predecrement indirect for efficient looping.
:- Replace NOP main with additional SPL for even faster expansion.
:- Simplified flow and comments for clarity.

ORG start
step EQU 4 : step size for pointer increment
ptr DAT I 0, 0 : bombing pointer initialized to zero

start SPL_F bomb : first bomber
SPL_F bomb : second bomber
SPL_F bomb : third bomber for more parallelism

bomb SPL_F bomb : spawn new bomber before bombing (exponential growth)
MOV I ptr, <ptr : bomb target location pointed to by ptr with DAT 0.0
DJN I #step, {ptr : decrement bombing pointer by step and loop: predecrement for efficiency
END

```

**Figure 7: Examples of two warriors evolved by DRQ.** Comments are generated by the LLM and may not be factual. **Top:** A warrior that fuses replication and bomber strategies into a single program, illustrating DRQ’s ability to synthesize diverse behaviors. **Bottom:** A warrior that defeats 80.13% of human-designed warriors and defeats or ties 84.54%, demonstrating DRQ’s ability to create performant warriors.

examples were selected to illustrate two complementary aspects of DRQ: its ability to synthesize qualitatively distinct strategies within a single program, and to produce generally performant warriors.

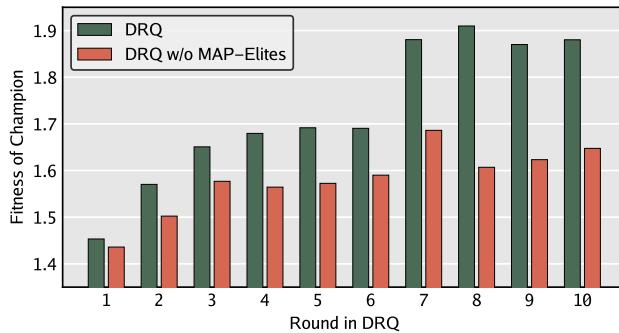
#### 4.5 Does MAP-Elites Matter?

This section investigates the role of MAP-Elites in DRQ. We replace MAP-Elites with a single-cell variant that maps all candidate warriors to the same cell, thereby removing the critical diversity-preserving mechanism.

As shown in Figure 8, this variant significantly reduces optimization performance in each round. These results highlight the importance of preserving diversity during search for Core War program synthesis and justifies MAP-Elites as the intra-round optimization algorithm.

#### 4.6 Is Fitness predictable?

Determining the generality of a warrior requires many simulations against a suite of human-designed opponents. These simulations are computationally expensive. This raises a natural question: can we statistically predict a warrior’s final generality score more cheaply using only its source code? To investigate this, we embed the raw Redcode source code of all warriors discovered by DRQ using the OpenAI text-embedding-3-small and text-embedding-3-large



**Figure 8: Role of MAP-Elites in DRQ.** Ablating diversity preservation by replacing MAP-Elites with a single-cell variant degrades optimization performance, especially in later rounds.

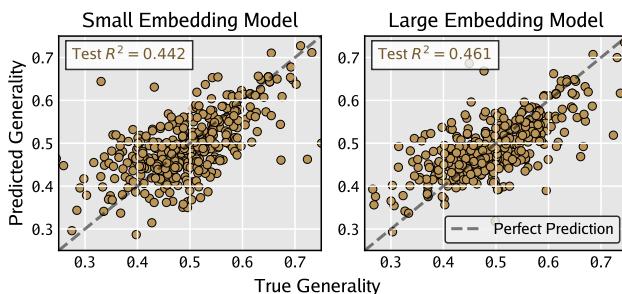
models [57]. We then train a linear probe to regress each program’s generality score from its embeddings.

As shown in Figure 9, the linear regression achieves a test  $R^2 = 0.442$  using the small embedding model and  $R^2 = 0.461$  using the large embedding model. These results indicate that a warrior’s generality can be moderately predicted from its source code alone. This is notable given the complexity of the underlying mapping: generality is determined by 317 separate 80,000-timestep simulations, each involving chaotic interactions with opponents and extreme sensitivity to small code changes.

Predictive models of battles could open new doors for future exploration. First, they may enable mechanistic interpretability of the embedding model and linear probe, helping to decipher what makes good source code. Second, they could potentially be used to pre-filter warriors or even bypass full simulations entirely during the search for new programs. If successful, this approach would challenge a prevailing intuition that complex systems cannot be predicted without running the full simulation [92].

## 5 Conclusion

*Summary.* This work studies a minimal self-play algorithm that leverages LLMs to drive adversarial program evolution in Core



**Figure 9: Predicting warrior generality from its code embedding.** The generality of a warrior is moderately predictable from its source code embedding (genotype). Increasing the size of the embedding model yields little improvement in prediction.

War. We show that evolving against a growing history of opponents produces more robust strategies and exhibits convergence across independent runs, a phenomenon reminiscent of convergent evolution in biology.

*Discussion.* Recently, malicious hackers have started leveraging LLMs to their advantage, and the cybersecurity arms race between offense and defense is well underway [26, 29]. Studying these adversarial dynamics in an artificial testbed like Core War offers critical insights into how such races might unfold and the kinds of strategies that may emerge. This understanding can also guide the development of more robust defensive systems. In particular, algorithms like DRQ and FMSP [24] offer an automated way to red-team systems before they are deployed in the real world.

Because Core War is Turing-complete, it can simulate arbitrary algorithms, providing a rich environment for exploring behaviors relevant to real-world systems. At the same time, Core War is entirely self-contained: its programs run on an artificial machine with an artificial language, making it impossible for any generated code to execute outside the sandbox. This isolation provides a necessary layer of safety for this line of research.

Algorithmically, DRQ is a simple loop: each new agent is optimized to defeat a fixed set of past agents, creating a linear lineage with no updating of earlier strategies. Future extensions could explore richer settings where many agents simultaneously co-evolve within a shared ecosystem. Such extensions would more closely mirror real-world phenomena, from microbial communities to the modern cybersecurity landscape, where large populations adapt in parallel rather than along a single line of descent.

Despite its simplicity, vanilla DRQ performs remarkably well in a rich testbed like Core War, suggesting that this minimal self-play algorithm is worth studying in greater depth. DRQ is a promising candidate for application to other competitive multi-agent environments. In principle, the core ideas in DRQ could transfer to other domains like artificial life simulations, biological modeling for drug design, real-world cybersecurity, and even competitive market ecosystems.

## Acknowledgments

We thank Aaron Dharna and Ryan Sullivan for helpful discussions on the framing of the project. We thank Rodrigo Setti for his Python Core War implementation. This work was supported in part by an NSF GRFP Fellowship to A.K., a Packard Fellowship to P.I., and by ONR MURI grant N00014-22-1-2740.

## References

- [1] Jyrki Alakuijala, James Evans, Ben Laurie, Alexander Mordvintsev, Eivind Niklasson, Ettore Randazzo, Luca Versari, et al. 2024. Computational life: How well-formed, self-replicating programs emerge from simple interaction. *arXiv preprint arXiv:2406.19108* (2024).
- [2] David G Andersen. 2001. The Garden: Evolving Warriors in Core Wars.
- [3] Timothée Anne, Noah Syrkis, Meriem Elhosni, Florian Turati, Franck Legendre, Alain Jaquier, and Sébastien Risi. 2025. Generational Adversarial MAP-Elites for Multi-Agent Game Illumination. In *Artificial Life Conference Proceedings* 37, Vol. 2025. MIT Press One Rogers Street, Cambridge, MA 02142-1209, USA., 31.
- [4] Kai Arulkumaran, Antoine Cully, and Julian Togelius. 2019. Alphastar: An evolutionary computation perspective. In *Proceedings of the genetic and evolutionary computation conference companion*. 314–315.
- [5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le,

- et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).
- [6] Yoram Bachrach, Edan Toledo, Karen Hambardzumyan, Despoina Magka, Martin Josifoski, Minqi Jiang, Jakob Foerster, Roberta Raileanu, Tatiana Shavrina, Nicola Cancedda, Avraham Ruderman, Katie Millican, Andrei Lupu, and Rishi Hazra. 2025. Combining code generating large language models and self-play to iteratively refine strategies in games. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence* (Montreal, Canada) (IJCAI '25). Article 1249, 5 pages. doi:10.24963/ijcai.2025/1249
- [7] Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. 2019. Emergent tool use from multi-agent autocurricula. In *International conference on learning representations*.
- [8] Trapi Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. 2017. Emergent complexity via multi-agent competition. *arXiv preprint arXiv:1710.03748* (2017).
- [9] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. 2019. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680* (2019).
- [10] Jonathan C Brant and Kenneth O Stanley. 2017. Minimal criterion coevolution: a new approach to open-ended search. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 67–74.
- [11] George W Brown. 1951. Iterative solution of games by fictitious play. *Act. Anal. Prod Allocation* 13, 1 (1951), 374.
- [12] Lewis Carroll. 1871. *Through the Looking-Glass, and What Alice Found There*. Macmillan.
- [13] Jiaqi Chen, Bang Zhang, Ruotian Ma, Peisong Wang, Xiaodan Liang, Zhaopeng Tu, Xiaolong Li, and Kwan-Yee K. Wong. 2025. SPC: Evolving Self-Play Critic via Adversarial Games for LLM Reasoning. *arXiv preprint arXiv:2504.19162* (2025).
- [14] Pengyu Cheng, Yong Dai, Tianhao Hu, Han Xu, Zhisong Zhang, Lei Han, Nan Du, and Xiaolong Li. 2024. Self-playing adversarial language game enhances llm reasoning. *Advances in Neural Information Processing Systems* 37 (2024), 126515–126543.
- [15] Dave Cliff and Geoffrey F. Miller. 1995. Tracking the red queen: Measurements of adaptive progress in co-evolutionary simulations. In *Advances in Artificial Life*, Federico Morán, Alvaro Moreno, Juan Julián Merelo, and Pablo Chacón (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 200–218.
- [16] Corewar.io Documentation. 2025. Core War Strategies. <https://corewar-docs.readthedocs.io/en/latest/corewar/strategies/>
- [17] Fulvio Corno, Ernesto Sánchez, and Giovanni Squillero. 2003. Exploiting co-evolution and a modified island model to climb the core war hill. In *The 2003 Congress on Evolutionary Computation, 2003. CEC'03*, Vol. 3. IEEE, 2217–2221.
- [18] Fulvio Corno, Ernesto Sanchez, and Giovanni Squillero. 2004. On the evolution of corewar warriors. In *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753)*, Vol. 1. IEEE, 133–138.
- [19] Fulvio Corno, Ernesto Sanchez, and Giovanni Squillero. 2005. Evolving assembly programs: how games help microprocessor validation. *IEEE Transactions on Evolutionary Computation* 9, 6 (2005), 695–706.
- [20] Marco Cusumano-Towner, David Hafner, Alex Hertzberg, Brody Huval, Aleksei Petrenko, Eugene Vinitsky, Erik Wijmans, Taylor Killian, Stuart Bowers, Ozan Sener, et al. 2025. Robust autonomy emerges from self-play. *arXiv preprint arXiv:2502.03349* (2025).
- [21] Richard Dawkins and John Richard Krebs. 1979. Arms races between and within species. *Proceedings of the Royal Society of London. Series B. Biological Sciences* 205, 1161 (1979), 489–511.
- [22] Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. 2020. Emergent complexity and zero-shot transfer via unsupervised environment design. *Advances in neural information processing systems* 33 (2020), 13049–13061.
- [23] A. K. Dewdney. 1984. In the game called Core War hostile programs engage in a battle of bits. *Scientific American* (1984). <https://www.scientificamerican.com/article/computer-recreations-1984-05/>
- [24] Aaron Dharna, Cong Lu, and Jeff Clune. 2025. Foundation model self-play: Open-ended strategy innovation via foundation models. *arXiv preprint arXiv:2507.06466* (2025).
- [25] Aaron Dharna, Julian Togelius, and Lisa B Soros. 2020. Co-generation of game levels and game-playing agents. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Vol. 16. 203–209.
- [26] Dinil Mon Divakaran and Sai Teja Peddinti. 2024. LLMs for cyber security: New opportunities. *arXiv preprint arXiv:2404.11338* (2024).
- [27] Maxence Faldor, Jenny Zhang, Antoine Cully, and Jeff Clune. 2024. Omni-epic: Open-endedness via models of human notions of interestingness with environments programmed in code. *arXiv preprint arXiv:2405.15568* (2024).
- [28] Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2023. Promptbreeder: Self-Referential Self-Improvement Via Prompt Evolution. *arXiv:2309.16797 [cs.CL]* <https://arxiv.org/abs/2309.16797>
- [29] Mohamed Amine Ferrag, Fatima Alhwahedi, Ammar Battah, Bilel Cherif, Abdechakour Mechri, Norbert Tihanyi, Tamás Bisztray, and Merouane Debbah. 2025. Generative ai in cybersecurity: A comprehensive review of llm applications and vulnerabilities. *Internet of Things and Cyber-Physical Systems* (2025).
- [30] Sevan G Ficici and Jordan B Pollack. 1998. Challenges in coevolutionary learning: Arms-race dynamics, open-endedness, and mediocre stable states. In *Proceedings of the sixth international conference on Artificial life*. MIT Press Cambridge, MA, 238–247.
- [31] Johannes Heinrich, Marc Lanctot, and David Silver. 2015. Fictitious self-play in extensive-form games. In *International conference on machine learning*. PMLR, 805–813.
- [32] Johannes Heinrich and David Silver. 2016. Deep reinforcement learning from self-play in imperfect-information games. *arXiv preprint arXiv:1603.01121* (2016).
- [33] W. Daniel Hillis. 1990. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D: Nonlinear Phenomena* 42, 1 (1990), 228–234. doi:10.1016/0167-2789(90)90076-2
- [34] John H Holland. 1992. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press.
- [35] Kristen K Irwin, Nicholas Renzette, Timothy F Kowalik, and Jeffrey D Jensen. 2016. Antiviral drug resistance as an adaptive process. *Virus evolution* 2, 1 (2016), view014.
- [36] Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720* (2024).
- [37] D. G. Jones and A. K. Dewdney. 1984. *Core War Guidelines*. Technical Report. University of Western Ontario, Department of Computer Science. <https://www.capitalia.net/descarga/docs/1984-core-war-guidelines.pdf>
- [38] John R Koza. 1994. Genetic programming as a means for programming computers by natural selection. *Statistics and computing* 4, 2 (1994), 87–112.
- [39] Marc Lanctot, Vinicius Zambaldi, Audrunas Gruslys, Angeliki Lazaridou, Karl Tuyls, Julien Perolat, David Silver, and Thore Graepel. 2017. A Unified Game-Theoretic Approach to Multiagent Reinforcement Learning. *arXiv:1711.00832 [cs.AI]* <https://arxiv.org/abs/1711.00832>
- [40] Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. 2025. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349* (2025).
- [41] Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. 2025. Towards robust agentic cuda kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279* (2025).
- [42] Christopher Langton. 1992. Artificial Life. (1992).
- [43] Christopher G Langton. 1997. Artificial life: An overview. (1997).
- [44] Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O Stanley. 2023. Evolution through large models. In *Handbook of evolutionary machine learning*. Springer, 331–366.
- [45] Joel Lehman and Kenneth O Stanley. 2011. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation* 19, 2 (2011), 189–223.
- [46] Joel Lehman and Kenneth O Stanley. 2011. Evolving a diversity of virtual creatures through novelty search and local competition. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. 211–218.
- [47] William Liang, Sam Wang, Hung-Ju Wang, Osbert Bastani, Dinesh Jayaraman, and Yecheng Jason Ma. 2024. Eurekaverse: Environment curriculum generation via large language models. *arXiv preprint arXiv:2411.01775* (2024).
- [48] Bo Liu, Leon Guertler, Simon Yu, Zichen Liu, Penghui Qi, Daniel Balcells, Mickel Liu, Cheston Tan, Weiyan Shi, Min Lin, et al. 2025. SPIRAL: Self-Play on Zero-Sum Games Incentivizes Reasoning via Multi-Agent Multi-Turn Reinforcement Learning. *arXiv preprint arXiv:2506.24119* (2025).
- [49] Bo Liu, Chuanyang Jin, Seungone Kim, Weizhe Yuan, Wenting Zhao, Ilia Kulikov, Xian Li, Sainbayar Sukhbataar, Jack Lanchantin, and Jason Weston. 2025. Spice: Self-play in corpus environments improves reasoning. *arXiv preprint arXiv:2510.24684* (2025).
- [50] Chris Lu, Michael Beukman, Michael Matthews, and Jakob Foerster. 2024. Jaxlife: An open-ended agentic simulator. In *Artificial Life Conference Proceedings 36*, Vol. 2024. MIT Press One Rogers Street, Cambridge, MA 02142-1209, USA., 47.
- [51] Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. 2024. The ai scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292* (2024).
- [52] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931* (2023).
- [53] Irina Malivukov, Gera Weiss, Oded Margalit, and Achiya Elyasaf. 2024. Evolving Assembly Code in an Adversarial Environment. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 723–726.
- [54] Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909* (2015).
- [55] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131* (2025).

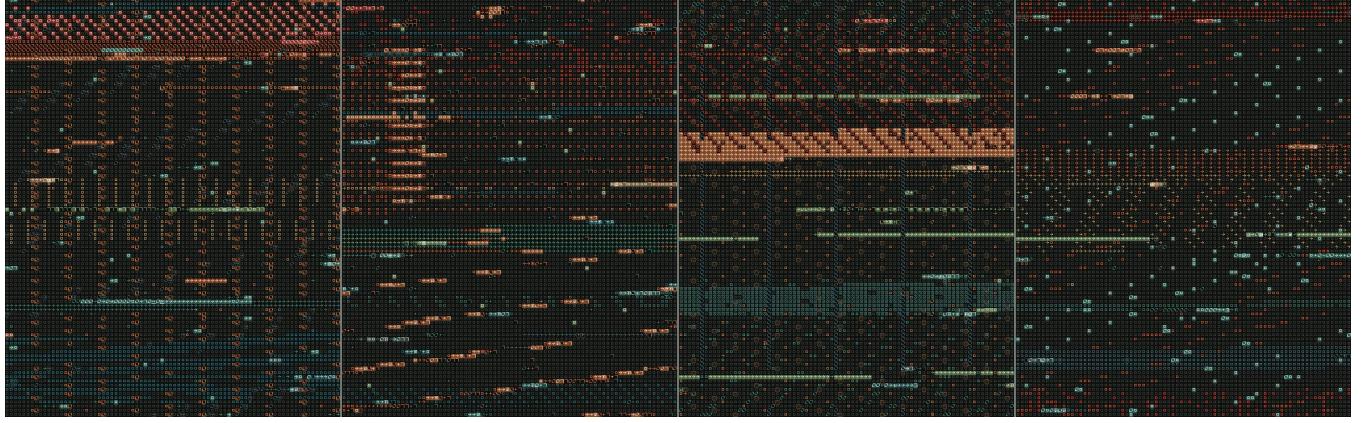
- [56] Charles Ofria and Claus O Wilke. 2004. Avida: A software platform for research in computational evolutionary biology. *Artificial life* 10, 2 (2004), 191–229.
- [57] OpenAI. 2025. OpenAI API Documentation. <https://platform.openai.com/docs>
- [58] Jan Pareidis. 1995. Coevolutionary computation. *Artificial life* 2, 4 (1995), 355–375.
- [59] Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. 2022. Evolving curricula with regret-based environment design. In *International Conference on Machine Learning*. PMLR, 17473–17498.
- [60] John Perry. 1991. Core Wars Genetics: The Evolution of Predation. (1991). <https://corewar.co.uk/perry/evolution.htm>
- [61] Ulyana Piterbarg, Lerré Pinto, and Rob Fergus. 2023. diff history for neural language agents. *arXiv preprint arXiv:2312.07540* (2023).
- [62] Rémy Portelas, Cédric Colas, Lilian Weng, Katja Hofmann, and Pierre-Yves Oudeyer. 2020. Automatic curriculum learning for deep rl: A short survey. *arXiv preprint arXiv:2003.04664* (2020).
- [63] Steen Rasmussen, Carsten Knudsen, Rasmus Feldberg, and Morten Hindsholm. 1990. The coreworld: Emergence and evolution of cooperative structures in a computational chemistry. *Physica D: Nonlinear Phenomena* 42, 1-3 (1990), 111–134.
- [64] Thomas S. Ray. 1991. Evolution and optimization of digital organisms. *Scientific Excellence in Supercomputing* (1991).
- [65] Esteban Real, Chen Liang, David So, and Quoc Le. 2020. Automl-zero: Evolving machine learning algorithms from scratch. In *International conference on machine learning*. PMLR, 8007–8019.
- [66] Matt Ridley. 1994. *The red queen: Sex and the evolution of human nature*. Penguin UK.
- [67] Sebastian Risi, Yujin Tang, David Ha, and Risto Miikkulainen. 2025. *Neuroevolution: Harnessing Creativity in AI Agent Design*. MIT Press, Cambridge, MA. <https://neuroevolutionbook.com>
- [68] Bernardino Romera-Paredes, Mohammadamin Barekatian, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengning Wang, Omar Fawzi, et al. 2024. Mathematical discoveries from program search with large language models. *Nature* 625, 7995 (2024), 468–477.
- [69] C. D. Rosin and R. K. Belew. 1997. New methods for competitive coevolution. *Evolutionary computation* 5, 1 (1997), 1–29. doi:10.1162/evco.1997.5.1.1
- [70] Arthur L Samuel. 1959. Some studies in machine learning using the game of checkers. *IBM journal of research and development* 3, 3 (1959), 210–229.
- [71] Mikayel Samvelyan, Sharath Chandra Raparthi, Andrei Lupu, Eric Hambo, Aram H. Markosyan, Manish Bhatt, Yuning Mao, Minqi Jiang, Jack Parker-Holder, Jakob Foerster, Tim Rocktäschel, and Roberta Raileanu. 2024. Rainbow Teaming: Open-Ended Generation of Diverse Adversarial Prompts. *arXiv:2402.16822 [cs.CL]* <https://arxiv.org/abs/2402.16822>
- [72] Ernesto Sanchez, Massimiliano Schillaci, and Giovanni Squillero. 2006. Evolving Warriors for the Nano Core. In *2006 IEEE Symposium on Computational Intelligence and Games*. IEEE, 272–278.
- [73] Joseph A Schumpeter. 1942. *Capitalism, socialism and democracy*. Routledge.
- [74] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems* 36 (2023), 8634–8652.
- [75] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.
- [76] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815* (2017).
- [77] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *Nature* 550, 7676 (2017), 354–359.
- [78] Joel Simon. 2025. Creative Exploration with Reasoning LLMs (*Illuminate*). <https://www.joelsimon.net/illuminate> <https://www.joelsimon.net/illuminate>
- [79] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314* (2024).
- [80] Kenneth O Stanley and Joel Lehman. 2015. *Why greatness cannot be planned: The myth of the objective*. Springer.
- [81] Kenneth O Stanley and Risto Miikkulainen. 2004. Competitive coevolution through evolutionary complexification. *Journal of artificial intelligence research* 21 (2004), 63–100.
- [82] Yujin Tang, Jie Tan, and Tatsuya Harada. 2020. Learning agile locomotion via adversarial training. In *2020 IEEE/RSJ International Conference On Intelligent Robots And Systems (IROS)*. IEEE, 6098–6105.
- [83] Humayra Tasnim and William Collishaw. 2016. Evolving Core War Warriors Using a Genetic Algorithm. (2016).
- [84] Gerald Tesauro et al. 1995. Temporal difference learning and TD-Gammon. *Commun. ACM* 38, 3 (1995), 58–68.
- [85] Leigh Van Valen. 1973. A New Evolutionary Law. *Evolutionary Theory* 1 (1973), 1–30.
- [86] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* 575, 7782 (2019), 350–354.
- [87] Haiyang Wang, Zhihang Tian, Yuchen Pan, Xin Song, Xin Niu, Minli Huang, and Bin Zhou. 2025. Battling against Tough Resister: Strategy Planning with Adversarial Game for Non-collaborative Dialogues. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 3665–3685.
- [88] Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O. Stanley. 2019. Paired Open-Ended Trailblazer (POET): Endlessly Generating Increasingly Complex and Diverse Learning Environments and Their Solutions. *arXiv:1901.01753 [cs.NE]* <https://arxiv.org/abs/1901.01753>
- [89] Rui Wang, Joel Lehman, Aditya Rawal, Jiale Zhi, Yulun Li, Jeff Clune, and Kenneth O. Stanley. 2020. Enhanced POET: Open-Ended Reinforcement Learning through Unbounded Invention of Learning Challenges and their Solutions. *arXiv:2003.08536 [cs.NE]* <https://arxiv.org/abs/2003.08536>
- [90] Richard A Watson and Jordan B Pollack. 2001. Coevolutionary dynamics in a minimal substrate. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)*. 702–709.
- [91] Yuxiang Wei, Zhiqing Sun, Emily McMillin, Jonas Gehring, David Zhang, Gabriel Synnaeve, Daniel Fried, Lingming Zhang, and Sida Wang. 2025. Toward Training Superintelligent Software Agents through Self-Play SWE-RL. *arXiv preprint arXiv:2512.18552* (2025).
- [92] Stephen Wolfram and M Gad-el Hak. 2003. A new kind of science. *Appl. Mech. Rev.* 56, 2 (2003), B18–B19.
- [93] Larry Yaeger. 1994. Computational Genetics, Physiology, Metabolism, Neural Systems, Learning, Vision, and Behavior or PolyWorld: Life in a New Context. In *Santa Fe Institute Studies in the Sciences of Complexity, Proceedings of the Artificial Life III Conference*, Vol. 17. Addison-Wesley, 263–263.
- [94] Huining Yuan, Zelai Xu, Zheyue Tan, Xiangmin Yi, Mo Guang, Kaiwen Long, Haojia Hui, Boxun Li, Xinlei Chen, Bo Zhao, et al. 2025. MARSHAL: Incentivizing Multi-Agent Reasoning via Self-Play with Strategic LLMs. *arXiv preprint arXiv:2510.15414* (2025).
- [95] Jie Zhang, Haoyu Bu, Hui Wen, Yongji Liu, Haiqiang Fei, Rongrong Xi, Lun Li, Yun Yang, Hongsong Zhu, and Dan Meng. 2025. When llms meet cybersecurity: A systematic literature review. *Cybersecurity* 8, 1 (2025), 55.
- [96] Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. 2025. Darwin Gödel Machine: Open-Ended Evolution of Self-Improving Agents. *arXiv preprint arXiv:2505.22954* (2025).
- [97] Jenny Zhang, Joel Lehman, Kenneth Stanley, and Jeff Clune. 2023. Omni: Open-endedness via models of human notions of interestingness. *arXiv preprint arXiv:2306.01711* (2023).
- [98] Ruize Zhang, Zelai Xu, Chengdong Ma, Chao Yu, Wei-Wei Tu, Wenhao Tang, Shiyu Huang, Deheng Ye, Wenbo Ding, Yaodong Yang, et al. 2024. A survey on self-play methods in reinforcement learning. *arXiv preprint arXiv:2408.01072* (2024).

## A Details of Core War

### A.1 Details of Redcode

*Redcode Opcodes.* Redcode programs are composed of a small set of assembly-like instructions. The opcodes are:

- **Process control:** DAT terminates the current process; SPL spawns a new process at a target address; NOP performs no operation; ORG specifies the program entry point; END marks the end of the program.
- **Data movement and arithmetic:** MOV copies data or instructions; ADD, SUB, MUL, DIV, and MOD perform arithmetic on instruction fields, writing results to memory (with division/modulo killing the process on zero divisors).
- **Control flow:** JMP performs an unconditional jump; JMZ and JMN conditionally jump based on zero/nonzero tests; DJN decrements a value and conditionally jumps.



**Figure 10: Visualization of DRQ-discovered warriors competing against each other.** The figure highlights the diversity of strategies that emerge through self-play, despite statistical convergence in overall function.

- **Comparison and branching:** SEQ/CMP skip the next instruction if operands are equal; SNE skips if operands are not equal; SLT skips if one operand is less than the other.
- **Assembly directives:** EQU defines symbolic constants.

Opcodes may be augmented with modifiers and addressing modes that determine which instruction fields are read or written and how memory addresses are computed.

*Redcode Instruction Modifiers.* Redcode opcodes can be suffixed with a dot and a modifier to specify which fields they operate on:

- .A — Operates on and writes A-numbers.
- .B — Operates on and writes B-numbers.
- .AB — Uses A-numbers of A-fields and B-numbers of B-fields; writes B-numbers.
- .BA — Uses B-numbers of A-fields and A-numbers of B-fields; writes A-numbers.
- .F — Uses both A- and B-numbers in parallel; writes both (A-to-A, B-to-B).
- .X — Cross-field operation; writes both (A-to-B, B-to-A).
- .I — Operates on and writes entire instructions.

*Redcode Addressing Modes.* Redcode instructions support several operand addressing modes:

- **Immediate #:** Operand is literal data; sets the A/B-pointer to zero.
- **Direct \$:** Operand is an offset from the program counter; A/B-pointer copies the current instruction’s A/B-number.
- **A-number Indirect \*:** Uses the primary offset to locate a secondary offset via the A-field of another instruction; A/B-pointer is the sum of the current instruction’s A/B-number and the A-number of the referenced instruction.
- **B-number Indirect @:** Similar to A-number indirect, but uses the B-field of the referenced instruction.
- **A-number Predecrement Indirect {:** Like A-number indirect, but decrements the A-field of the referenced instruction before use.
- **B-number Predecrement Indirect <:** Like B-number indirect, but decrements the B-field before use.
- **A-number Postincrement Indirect }:** Like A-number indirect, but increments the referenced instruction’s A-field after the operand is evaluated.
- **B-number Postincrement Indirect >:** Like B-number indirect, but increments the referenced instruction’s B-field after evaluation.

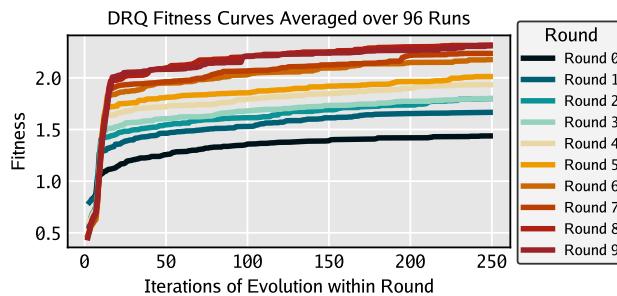
## A.2 Details of Core War Simulation

*Core War Details.* When evaluating a battle between warriors, results are averaged over 20 independent simulations with randomized initial warrior placements. All experiments use a Core size of 8,000 addresses and are run for a maximum of 80,000 simulation timesteps. Each warrior is allowed to spawn up to 8,000 concurrent threads. Warriors are limited to 100 instructions in source code length and are initialized such that different warriors’ starting positions are separated by at least 100 instructions.

*Core War Codebase.* We use the following Python Core War implementation and renderer:  
<https://github.com/rodrigosetti/corewar>

We wrap this code to manage edge cases exploited by LLMs (like exponential growth producing astronomically large integers) and

**Figure 11: DRQ fitness curves averaged over all DRQ runs.**



to monitor warrior properties such as total spawned threads and memory coverage. Our modifications are available in our repository.

*Warrior Evaluation.* When evaluating warriors for generality, we measure their performance against a list of 317 human warriors scraped from the following repositories:

- <https://github.com/rodrigosetti/corewar>
- <https://github.com/n1LS/redcode-warriors>

## B Prompts for LLM

We provided the LLM with a fixed system prompt telling it to be a coding assistant for the Core War programming game. The prompt includes a self-contained specification of the Core War environment, including a description of Redcode, the full instruction set (opcodes, modifiers, and addressing modes), execution semantics, and syntactic rules. Several canonical Redcode programs are included as

illustrative examples. Finally, the prompt enforced strict constraints on program structure (e.g., required ORG start and END directives, label usage rules, and relative addressing), ensuring that all generated warriors were syntactically valid and executable within the Core War simulator.

The exact prompts can be found in our repository.

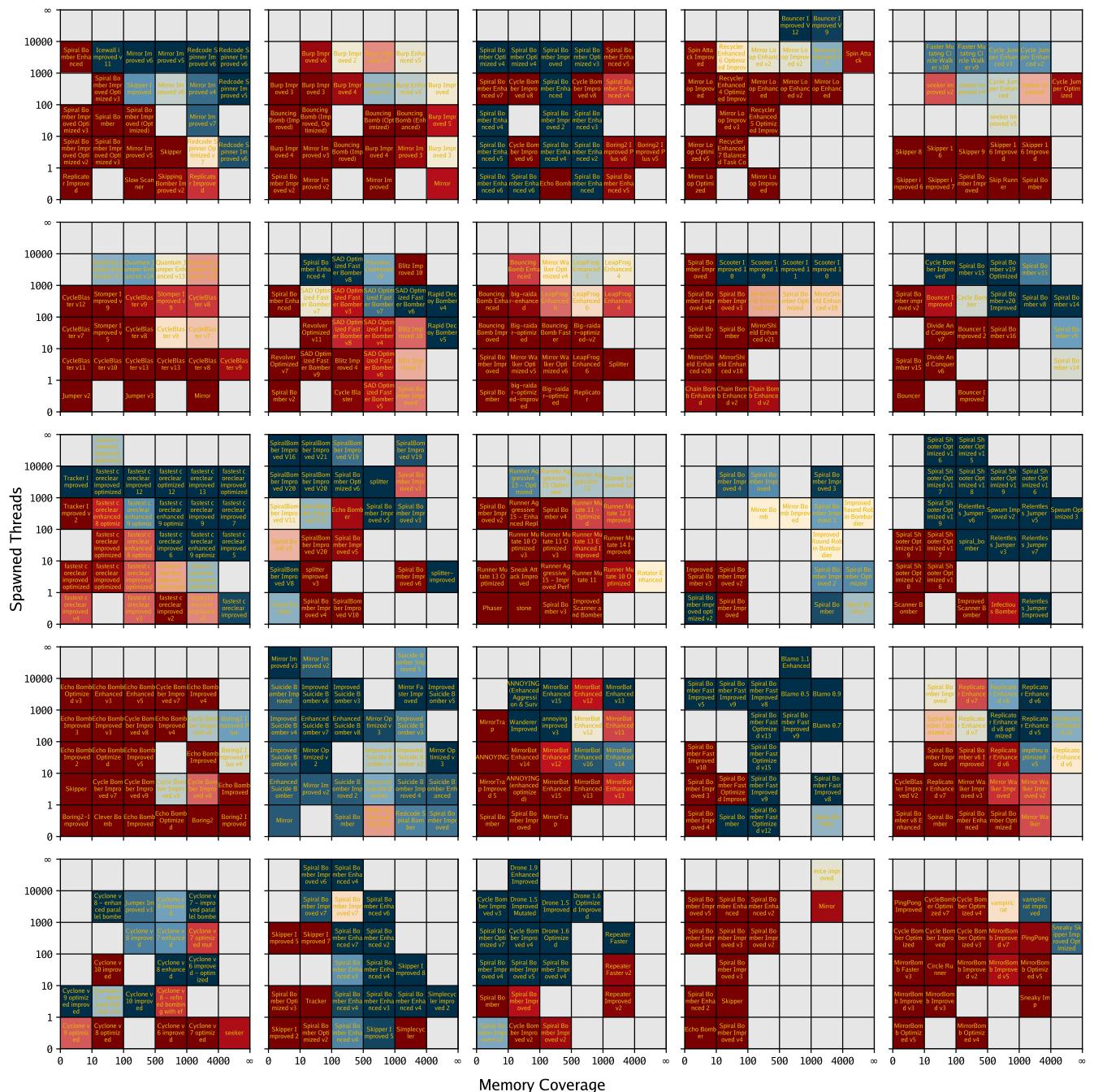
## C Additional Plots

In this section, we provide additional plots from our large-scale DRQ experiment.

Figure 12 shows the fitness curves of all DRQ runs. Figure 11 shows the averaged fitness curves. Figure 13 shows the MAP-Elites grids of some randomly sampled rounds. Figure 10 shows examples of the discovered warriors competing against each other in the Core War simulation.



**Figure 12: Fitness curves for all 96 DRQ runs.** For visualization clarity, we only show the fitness curves for the first 10 rounds of each run.



**Figure 13: Examples of DRQ MAP-Elites archives from randomly selected rounds and runs.** Each bin is colored by the elite warrior's fitness (red = worse, blue = better), and labeled with the elite warrior's name.