OPTIMIZING GENETIC PROGRAMMING AGENTS WITH TPG AND MEMORY STRUCTURES

OPTIMIZING GENETIC PROGRAMMING AGENTS WITH TPG AND MEMORY STRUCTURES

By TANYA DJAVAHERPOUR, BSc

A Thesis Submitted to the School of Graduate Studies in Partial Fulfillment of the Requirements for the Degree Master of Science

McMaster University

MASTER OF SCIENCE (2025)

Hamilton, Ontario, Canada (Computing and Software)

TITLE:	Optimizing Genetic Programming Agents with TPG and Memory Structures
AUTHOR:	Tanya Djavaherpour BSc (Computer Engineering), McMaster University, Hamilton, Ontario, Canada
SUPERVISOR:	Dr. Stephen Kelly
NUMBER OF PAGES:	xi, 64

Abstract

This thesis explores the design of temporal memory for Tangled Program Graphs (TPGs), a team-based Genetic Programming (GP) framework for Reinforcement Learning (RL). We specifically focus on challenging partially-observable settings in which agents rely on memory to handle temporal dependencies.

First, we look at how global indexed scalar memory can be initialized to better store and retrieve observations, helping agents build internal models of the environment. Tests on simple classic control tasks show that resetting memory at the beginning of each new interaction sequence with the environment can prevent interference by weaker agents and improve performance in tasks with shorter-term dependencies.

Next, we tackle partially-observable continuous control tasks with large state and action spaces. Here we propose team-specific shared memory, where each group of programs keeps its own memory without being affected by other teams. In addition, we extend TPG's scalar memory by adding vector and matrix structures initialized with evolved constants, which are numerical values that evolve across generations to give agents inherited knowledge. These enhancements allow for stronger coordination and more robust behaviour in high-dimensional tasks.

Overall, our findings highlight the vital role of indexed memory in TPGs when an agent lacks full state information. By exploring different ways to store and share

data among programs, this work highlights the importance of sharing information both among team members during the lifetime of an agent and across generations through evolved constants.

Acknowledgements

I would like to thank my supervisor, Dr. Stephen Kelly, for his generous guidance, thoughtful advice, and constant support throughout this work. His patience and insight not only helped me sharpen my research ideas, but also encouraged me to pursue new directions I had not initially considered. I am also grateful to Dr. Swati Mishra for her insightful feedback and encouraging discussions. I appreciate Dr. Matthew Giamou for facilitating the final steps of this journey.

I want to acknowledge the friendship and support of my friends all around the world, who cheered me on and kept me focused during the most challenging moments. Finally, I owe the biggest thanks to my family, especially my parents, as well as dear Mahsa and Sam, whose love and support carried me through from start to finish.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Foundations of Tangled Program Graphs

Evolutionary Computing (EC) is a family of population-based search and optimization algorithms inspired by the principles of natural selection and genetics. These algorithms evolve a set of candidate solutions over generations, favouring those that perform best. Performance is evaluated using a fitness function, which is defined specifically for each task [7].

Genetic Programming (GP), a key approach in EC, evolves computer programs to solve specific tasks. GP individuals are typically represented as executable structures such as trees [17] and linear instruction sequences [2], supporting the evolution of flexible solutions without fixed length. This makes GP particularly effective in domains where the structure of a solution is not known in advance, and where transparency and modularity are desired features.

In the context of RL, GP has shown promise in evolving agents. By interacting with the environment through trial and error, RL agents can learn to perform complex behaviours. Indexed memory enables GP-based agents to store and recall relevant information, while programmatic modularity supports structured and interpretable agents. This supports decision-making even under partial observability, when agents do not have access to the full state of their environment [2].

TPG is a GP framework designed to evolve interpretable and modular policies for RL tasks. In TPGs, agents are composed of teams of linear programs, where each program processes input data, interacts with its own register memory, and proposes an action. Each action comes along with a confidence value, which is called the bid value, and an action value. These programs compete through a bidding process, and the winning program determines the agent's action [13].

Adding indexed memory to TPGs functions as a "Culture" [24], empowering agents to share information. The global indexed memory, which is shared across all agents, without resetting supports both long-term and short-term information retrieval. Since the entire population has access to this memory, even poorly performing agents can write data to it, which negatively affects overall performance [21]. In this work, we explore different memory management strategies and memory structures in TPGs to improve their performance.

The rest of this thesis is organized as follows. Chapter 2 looks at memory management strategies, focusing on how resetting the global memory in tasks with short-term dependencies affects performance. By resetting the memory at the beginning of each RL trial, access is limited to the running agent, meaning that only its programs interact with the shared memory. Compared to the case where all agents' programs

interact with a global memory and forming a culture, this setup resembles a smaller and more focused society. Our results show that this approach leads to better outcomes.

Based on the findings in Chapter 2, Chapter 3 implements a team-specific shared memory. In this work, at the start of each RL trial, shared memory and programs' registers are initialized with constant values that evolve over generations. Team-specific shared memory allows teams to coordinate more effectively by sharing task-relevant information without interference from poorly performing agents. Evolved constants provide agents with inherited knowledge, so they do not begin with registers and indexed memory filled only with zeros. As a result, the study suggests that combining team-specific shared memory with evolved constants can improve TPGs agents' performance.

Finally, in Chapter 4, we summarize our findings and discuss possible directions for future work.

1.2 Publications Originating from this Thesis

Chapter 2 is adapted from our paper, "Tangled Program Graphs with Indexed Memory in Control Tasks with Short Time Dependencies," co-authored with Ali Naqvi and Stephen Kelly, and published in the *16th International Conference on Evolutionary Computation Theory and Applications (ECTA).*

Chapter 3 is adapted from our upcoming paper, "Genetic Encoding and Shared Knowledge in Reinforcement Learning with Structured Memory," co-authored with Stephen Kelly and submitted to the *ALIFE 2025 Conference.*

During my master's, I also contributed to a book chapter titled "Evolving Many-Model Agents with Vector and Matrix Operations in Tangled Program Graphs," co-authored with Ali Naqvi, Eddie Zhuang, and Stephen Kelly. This chapter appears in the book *Genetic Programming Theory and Practice XXI* (Springer Nature Singapore, 2025), and focuses on expanding TPGs by incorporating vector and matrix memory, enabling the evolution of multitask agents capable of handling control and forecasting problems in partially observable environments.

In addition, I contributed to the development of MAPLE: *Multi-Action Programs through Linear Evolution for Continuous Multi-Action Reinforcement Learning*, a collaborative work with Quentin Vacher, Stephen Kelly, Ali Naqvi, Nicolas Beuve, Mickaël Dardaillon, and Karol Desnos, accepted to the *Genetic and Evolutionary Computation Conference (GECCO) 2025*.

Chapter 2

Tangled Program Graphs with Indexed Memory in Control Tasks with Short Time Dependencies

2.1 Abstract

This paper addresses the challenges of shared temporal memory for evolutionary reinforcement learning agents in partially observable control tasks with short time dependencies. Tangled Program Graphs (TPG) is a genetic programming framework which has been widely studied in memory intensive tasks from video games, time series forecasting, and predictive control domains. In this study, we aim to improve external indexed memory usage in TPG by minimizing the impact of destructive agents during cultural transmission. We test various memory resetting strategies—per agent, per

episode, and a no-memory control group—and evaluate their effectiveness in mitigating destructive effects while maintaining performance. Results from Acrobot, Pendulum, and CartPole tasks show that resetting memory more often can significantly boost TPG performance while preserving computational efficiency. These findings highlight the importance of memory management in Reinforcement Learning (RL) and suggest opportunities for further optimization for more complex visual RL environments, including adaptive memory resetting and evolved probabilistic memory operations.

2.2 Introduction

Reinforcement Learning (RL) agents learn through trial-and-error interaction with their environment [26]. Deep Reinforcement Learning (DRL), with its capacity to decompose sensor inputs and build hierarchical representations of sensor data, has significantly expanded the capabilities of autonomous agents operating within complex environments [18]. Despite these advancements, DRL agents often encounter formidable obstacles in tasks necessitating robust memory functionalities [19]. This paper investigates these challenges and proposes simple strategies to enhance temporal memory capabilities in the recently-proposed genetic programming framework known as Tangled Program Graphs (TPG) [13].

Effective memory management is crucial for ensuring that agents can retain and utilize relevant information over time, particularly in environments that are only partially observable or which require long term planning. We explore various strategies

for enhancing the efficiency of indexed memory in TPG, with the goal of minimizing the negative impact of destructive agents and improving overall system performance. Through a series of experiments, we evaluate different memory management approaches, including probabilistic methods for writing into memory shared among a population of agents, and investigate their impact on the performance of TPG agents in partially observable benchmark RL environments with short time dependencies. This study focuses on comparing memory management strategies within TPG, using the original version of PyTPG [1] as the baseline. Our results demonstrate that clearing shared temporal memory before each evaluation episode improves agent performance by reducing the negative impact of destructive agents and lowering decision-making complexity.

2.3 Background

Genetic Programming (GP) is an Evolutionary Computation paradigm that evolves computer programs using evolutionary algorithms [2]. RL agents evolved with GP can model their environment over time through the use of temporal memory. In Linear Genetic Programming (LGP) [2], programs are represented by a sequence of instructions which read and write from memory registers. LGP supports a simple form of recursive temporal memory simply by allowing registers to maintain state between sequential program executions. More generally, GP can support *indexed memory* by augmenting agents with a linear memory array and adding specialized read and write operations to the GP function set [29]. If indexed memory is shared among agents in a population, it can also support the transmission of information between individuals by non-genetic means. Spector's [24], "Culture" allows all individuals to share the

same memory, similar to societal interactions, where each individual is affected by others in a shared environment, but risks "pollution" of the memory matrix by agents that perform badly.

In Visual RL, observable states are high-dimensional matrices such as video frames. TPG can directly process high-dimensional video inputs and has been tested in various gaming scenarios, outperforming traditional deep neural network RL methods in multi-task learning [13]. These TPG agents were also more computationally efficient, requiring fewer calculations per action than other approaches. Their efficiency is primarily due to: 1) the hierarchical complexity of each entity evolving based on its interaction with the problem domain, unlike the fixed complexity in conventional Deep Learning [18]; and 2) within a TPG entity, subsystems often focus on different segments of the visual input, meaning only certain components are active at any specific moment [14].

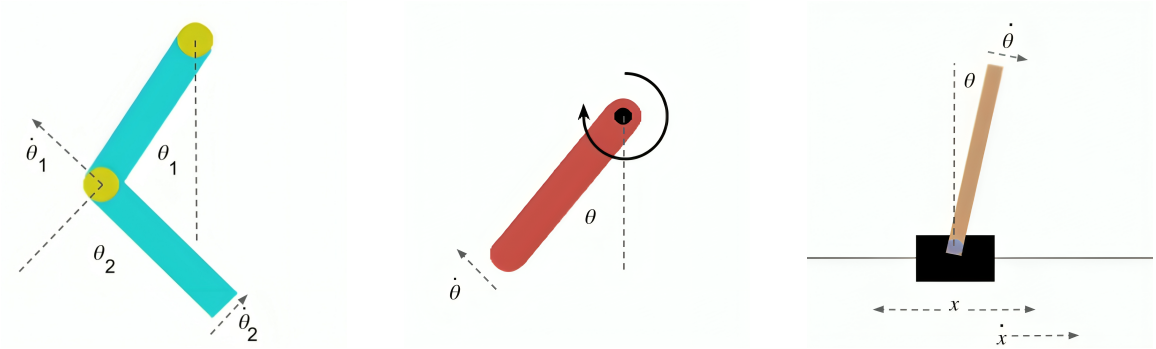
Despite visual RL providing high-resolution input, individual frames often lack the complete information required to select the best action. This *partial observability* significantly limits the agent's perception of the environment and implies that temporal memory must be available for the agent to build a mental model of its environment. TPG has successfully used emergent modularity combined with register memory *and* indexed memory to evolve problem solvers for memory-intensive tasks [15]. In short, TPG agents are composed of teams of linear genetic programs which share a single memory data structure and cooperatively manage a model of the environment which enables operation in partially observable RL tasks. In this work, we aim to enhance the effectiveness of indexed memory usage by minimizing the effects of destructive individuals during the cultural transmission of information through shared memory,

advancing our understanding of the "culture" of digital organisms.

2.4 Methodology

2.4.1 Environments

The environments used in this work are partially-observable versions of the widely-studied RL benchmarks Acrobot, Pendulum, and Cartpole [26], Figure 2.1. These tasks are selected for their high level of challenge, extensive comparative results available in the literature, and computational simplicity resulting in fast experiments.



(a) Acrobot (b) Pendulum (c) Cartpole

Figure 2.1: Problem environments used in this work. See [3] for details.

Acrobot

The Acrobot task is a dynamical system involving a double pendulum with 6 observation variables, indicated in Table 2.1, and 500 time steps. The control task involves swinging up the lower link of the double pendulum to reach a specified target height. As shown in Figure 2.1, the state of the Acrobot at every time step is given by the cosine and sine of the angles of the two links in radians (θ_1, θ_2) and their angular

velocities. The action space is discrete and consists of three actions: applying +1 torque, -1 torque, or no torque (0) to the second joint.

Table 2.1: Acrobot observation space

Num	Observation	Min	Max
0	$Cos(\theta_1)$	-1	1
1	$Sin(\theta_1)$	-1	1
2	$Cos(\theta_2)$	-1	1
3	$Sin(\theta_2)$	-1	1
4	θ_1 Angular Velocity	-4π	4π
5	θ_2 Angular Velocity	-9π	9π

The reward function is $-t_{end}$, which is reached when the free end hits the target height ($-\cos(\theta_1) - \cos(\theta_2 + \theta_1) > 1.0$) or when the episode exceeds 500 steps. The goal is to reach the target in as few steps as possible, with each step incurring a -1 reward, and reaching the target ending with a reward of 0.

Pendulum

The Pendulum task, shown in Figure 2.1, is a control problem with 3 observation variables and 200 time steps. This task involves swinging up a pendulum to an upright position and keeping it balanced. The action space consists of a single continuous control variable, representing the torque applied to the joint. The observation space consists of three elements which are indicated in Table 2.2.

Table 2.2: Pendulum observation space

Num	Observation	Min	Max
0	x = cos(θ)	-1.0	1.0
1	y = sin(θ)	-1.0	1.0
2	$\dot{\theta}$ = Angular Velocity	-0.8	0.8

The reward function is as follows:

$$\sum_{t=1}^{t_{max}} -(\phi(\theta)^2 + 0.1 \times \dot{\theta}^2 + 0.001 \times \text{Torque}^2) \tag{2.4.1}$$

In this reward function, $\phi(\theta)$ is the difference between the current angle θ and the upright position angle, and torque is the control input applied to the pendulum. The term $\phi(\theta)^2$ penalizes deviation from the upright position, $0.1 \times \dot{\theta}^2$ penalizes high angular velocities to encourage smoother movements, and $0.001 \times \text{Torque}^2$ penalizes large control inputs to promote energy efficiency.

Cartpole

The Cartpole task involves balancing a pole on a cart by applying force to the cart to keep the pole upright. This task has 4 observation variables given by cart position (x), cart velocity ($\dot{x}$), pole angle (θ), and pole velocity at the tip ($\dot{\theta}$). As shown in Figure 2.1, the state of the Cartpole at every time step is given by the cart position and velocity, pole angle in radians (θ), and pole angular velocity. The action space is discrete and consists of two actions, which represent pushing the cart to the left or right. The observation space consists of four elements which are indicated in Table 2.3.

Table 2.3: Cartpole observation space

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle (θ)	-0.418 rad	0.418 rad
3	Pole Angular Velocity	-Inf	Inf

The reward function is t_{end}, with +1 awarded for each time step the pole remains

upright. t_{end} is reached when the pole falls, the cart moves out of bounds, or the max number of steps is reached. The goal is to maximize the number of time steps the pole stays upright.

In all tasks, agent training fitness is its mean reward over 20 episodes, where each episode begins with random initial conditions and ends with success, failure, or reaching a time constraint. Post-evolution, the single training champion is reloaded and evaluated in 100 test episodes with initial conditions not seen during training.

Velocity state variables describe how the system is changing over time. To make these environments partially observable, we remove velocity state variables from the observation space. In order to control the systems without this information, agents **must** use temporal memory to store sequential observations over time and integrate this data to predict the velocity of the system. Note that predicting system velocities only requires short-term memory.

2.4.2 Tangled Program Graphs

Tangled Program Graphs (TPG) is a hierarchical algorithm for evolving teams of programs. The basic building block in TPG is a team of programs (see Figure 2.2). Each team represents a stand-alone decision-making entity (agent) in this framework. Each program is a linear structure consisting of registers and instructions that operate on observation inputs and internal memory registers. Programs return two values: a bid value and an action value. Teams follow a first-placed sealed bid auction method where the highest bidding program at each timestep wins the right to decide the action. This action could be a discrete value (directional forces in Figure 2.2), continuous value (contents of scalar register s[1] in Figure 2.2), or a pointer to another

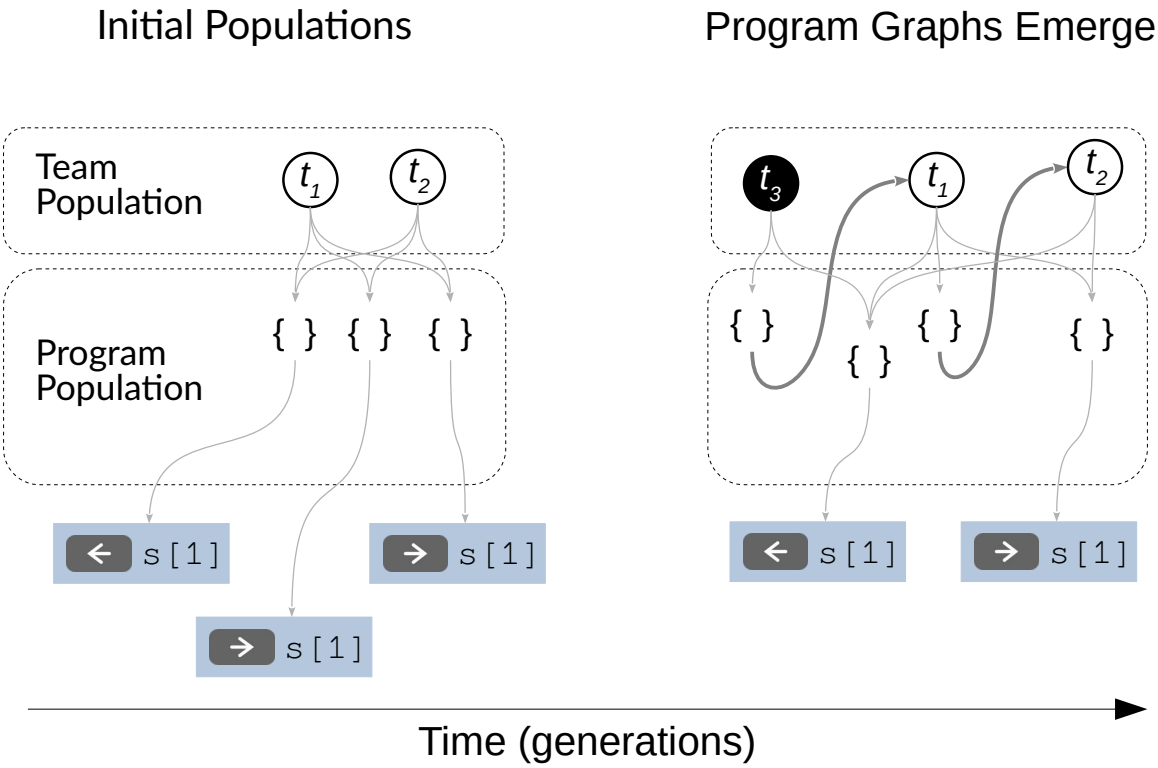


Figure 2.2: Tangled Program Graphs' hierarchical decision-making structure in which teams of programs predict discrete and continuous actions.

team. If the action is atomic (i.e. discrete or continuous) it is returned to the task environment as the control output for the current timestep. If the action is a team pointer, then decision-making is delegated and the bidding process repeats at this team for the same timestep and observation. The process repeats recursively until an atomic action is reached.

2.4.3 Memory

The TPG model introduced in [21] features an external shared memory accessible to all agents. Each agent consists of several teams and programs, and each program has its own private registers, which are inaccessible to other programs. Program registers

are stateful, and thus provide a simple form of recurrent temporal memory. Furthermore, all programs have access to the shared external memory for reading and writing operations. This memory is not reset between training episodes or the evaluation of different agents, ensuring continuity and allowing for cumulative knowledge building.

Indexed memory operations are handled probabilistically to manage both short-term and long-term retention. The write operations distribute the content of a program's registers across the external memory in a probabilistic manner, with locations in the middle of the memory being updated more frequently (short-term memory) and those towards the ends being updated less frequently (long-term memory). This study uses the following probability definition, which is shown in Figure 2.3, and where i corresponds to the index:

$$P_{\text{write}}(i) = \frac{0.25}{0.5\pi(i^2 + 0.25)} \tag{2.4.2}$$

This function provides a heavy-tailed distribution, allowing writing across a wide range of memory locations, the probability is sharply peaked at the center and rapidly decreases as the offset increases.

Read operations use indexing, allowing programs to locate regions of external memory characterized by specific temporal properties. This approach allows programs to interact during each generation or across different generations, facilitating more sophisticated decision-making strategies.

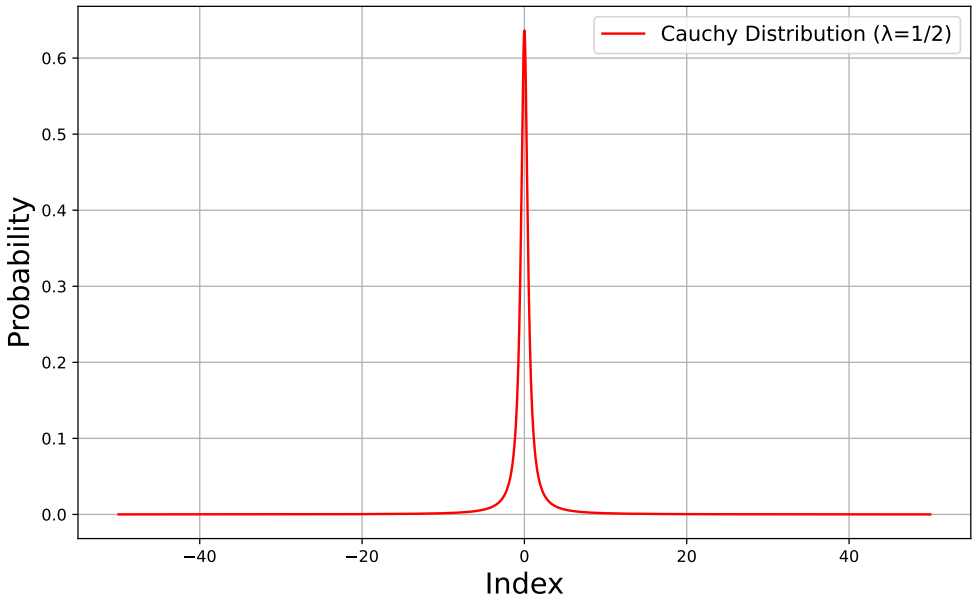


Figure 2.3: Probability function for memory write operations.

2.5 Experiments

The experiments detailed in this section are designed to evaluate our TPG shared temporal memory implementations in mitigating the negative impact of destructive agents while maintaining system performance and efficiency. We used TPG as implemented in [1]. The culture method discussed in [23] highlights the negative impact of destructive agents, noting that while positive ideas from individuals can be preserved for collective benefit, negative actions by a single agent can destroy valuable information. To reduce this negative impact, we study the effect of clearing memory at different stages and compare the results with the original version of shared memory in TPG.

Algorithm 1: Agent execution with memory resetting conditions.

```
for generation in generations do
    run agent with pooling
    if original_version then
        execute_episodes_with_frames()
    else
        if reset_for_each_agent then
            lock pooling
            reset external memory
            reset agent's registers
            execute_episodes_with_frames()
            release pooling
        else
            if reset_for_each_episode then
                for episode in episodes do
                    lock pooling
                    reset external memory
                    reset agent's registers
                    execute_frames()
                    release pooling
                end
            else
                for episode in episodes do
                    for frame in frames do
                        reset agent's registers
                        act and get feedback
                    end
                end
            end
        end
    end
end
Function execute_frames():
    for frame in frames do
        act and get feedback
    end
Function execute_episodes_with_frames():
    for episode in episodes do
        execute_frames();
    end
```

We assess three strategies: *resetting memory for each agent* (Section 2.5.1), *resetting memory for each episode* (Section 2.5.2), and a *no-memory* condition (Section 2.5.3). Algorithm 1 details the implementation of these strategies. The following terms are used in the pseudocode: `execute_frames()` executes a set of frames where the agent takes an action based on observations and receives feedback from the environment for each frame. **Lock pooling** and **release pooling** manage parallelism, with **lock pooling** preventing other agents from interacting with memory and **release pooling** restoring parallelism after the agent completes its interactions. `execute_episodes_with_frames()` runs multiple episodes.

2.5.1 Reset Memory for Each Agent

In this case, the external memory and registers are cleared and set to zero at the beginning of evaluating each agent in each generation. This method ensures that each agent can independently build its own memory model at run time and removes the possibility of negative impact from other agents. Each agent essentially has its own indexed memory which is shared among its programs, resembling a smaller society. In this case, the agent's memory maintains state over all training episodes, during which time the agent is free to gradually develop its mental model of the environment. While each agent interacts with memory, it is essential to restrict others' access to it. In our current implementation, this requires blocking the parallelizing system, which increases experiment run time.

2.5.2 Reset Memory for Each Episode

This approach also removes potential negative impact of other agents. In this version, we reset the external memory and all the agent's registers at the beginning of each episode. This tests the agents' ability to build their memory quickly during a single episode. Again, when one agent interacts with memory, it is essential to restrict others' access to it.

2.5.3 No Memory

In this version, we do not use any external indexed memory and we clear all the agent's registers to zero at the beginning of each time step, implying the agent's behaviour is entirely stateless. This is a control experiment to confirm that all partially observable task configurations absolutely require stateful agents with temporal memory capabilities.

2.5.4 Experimental Parameters

Evolutionary hyper-parameters follow previous TPG work in RL tasks [21]. The initial root team population is set at 360 and remains static throughout evolution. We utilize "Cauchy Half" (Equation 2.4.2) for memory distribution in scenarios involving memory. The operation set includes: "ADD", "SUB", "MULT", "DIV", "NEG", "COS", "LOG", "EXP", "MEM_READ", and "MEM_WRITE" allowing complex interactions without any task-specific functions. To constrain model complexity and computational cost of decision-making, we set the probability of acting atomic to be 1.0, meaning no programs point to another team.

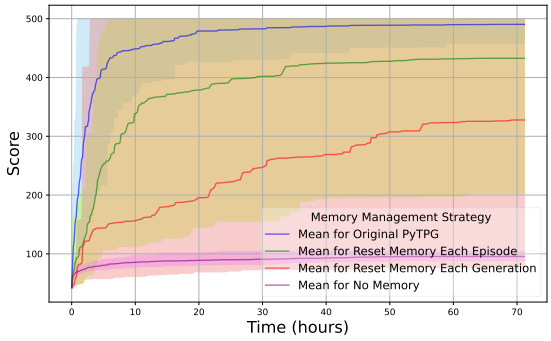
2.5.5 Results

Experiments reveal that the *reset memory for each episode* strategy (Section 2.5.2) improves the score and performance of TPG agents across all the control problems mentioned in Section 2.4.1, as shown in Figure 2.5. This memory configuration also results in the lowest solution complexity, as indicated in Figure 2.6.

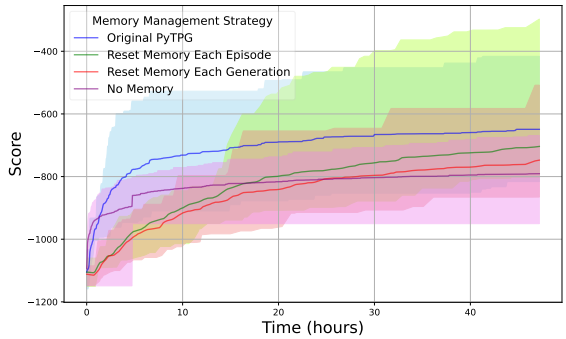
We conduct experiments for all the cases detailed in Section 2.5 as well as the original version of PyTPG [1], using the Cauchy Half distribution for memory writing probability. We ran 10 repeats with unique random seeds for Pendulum task and CartPole task, and 8 repeats for Acrobot task. Each experiment was run using multiple cores to manage the computational load efficiently: 30 hours with 30 cores for Acrobot, 48 hours with 10 cores for Pendulum, and 72 hours with 20 cores for CartPole. The results were compared based on the achieved score during the same running period (Figure 2.4), reached score based on the number of generations (Figure 2.5), and their complexity (Figure 2.6). The complexity is characterized by average number of instructions executed per action decision.

To plot Figures 2.5 and 2.4, we determined the minimum number of generations across all experiments. According to Figure 2.5, for all three environments, the approach of *resetting memory for each episode* has the best average score after the 5^{th} generation.

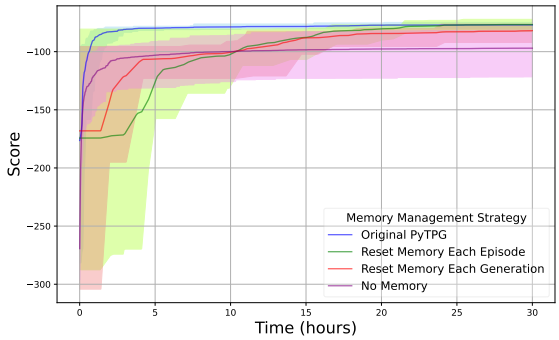
In Figure 2.6, the complexity over the minimum number of generations across all experiments is reported. This figure demonstrates that resetting memory and registers for each episode reduces complexity. Interestingly, in all three environments, although the *no memory* version has the worst score over generations, it exhibits the highest complexity. This indicates that agents are struggling to improve by making more



(a) Cartpole environment

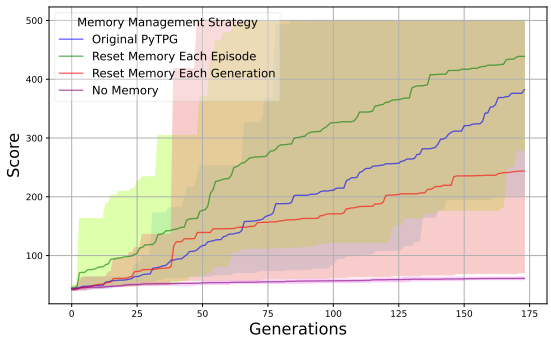


(b) Pendulum environment

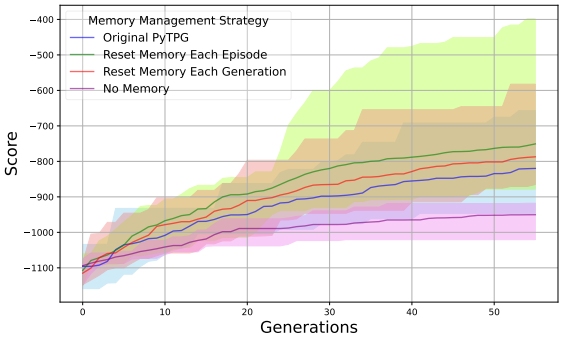


(c) Acrobot environment

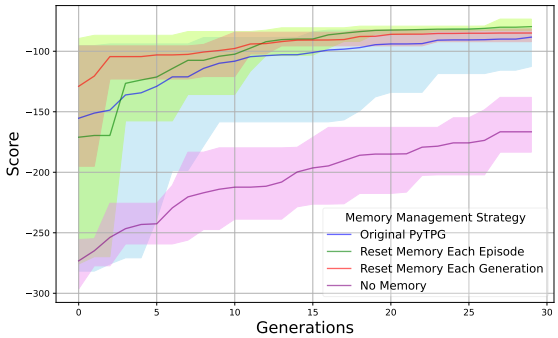
Figure 2.4: Scores achieved in different memory strategy experiments over 48 hours in each environment. Shaded areas show the range of scores across 10 repeats (minimum to maximum), and solid lines represent the average scores.



(a) Cartpole environment

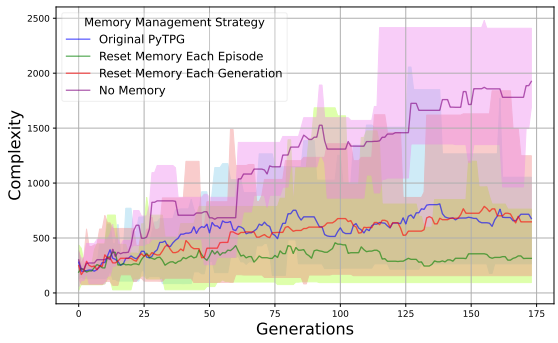


(b) Pendulum environment

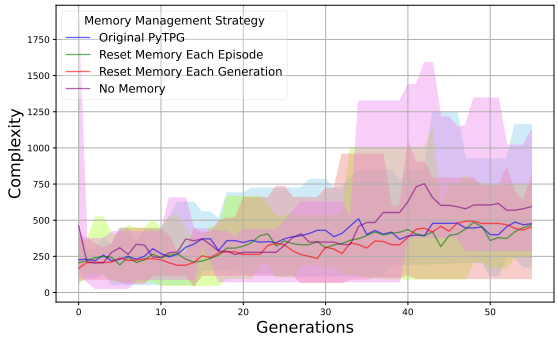


(c) Acrobot environment

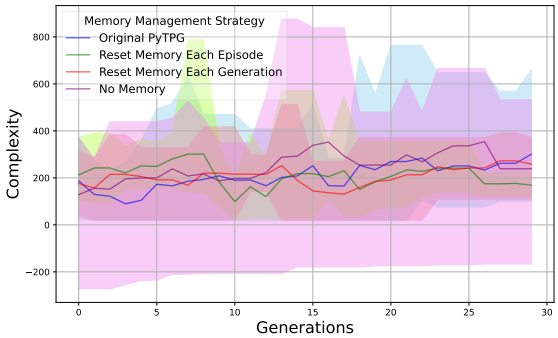
Figure 2.5: Scores achieved in different memory strategy experiments over 48 hours, based on the minimum number of generations run in various environments. Shaded areas show the range of scores across 10 repeats (minimum to maximum), and solid lines represent the average scores.



(a) Cartpole environment



(b) Pendulum environment



(c) Acrobot environment

Figure 2.6: Complexity in different memory strategy experiments over 48 hours, based on the minimum number of generations run in the different environment. Shaded areas show the range of complexity across 10 repeats (minimum to maximum), and solid lines represent the average scores.

complex decisions. On the other hand, the version with *resetting memory for each episode*, which has the highest score, also exhibits less computational complexity than the original PyTPG.

Execution speed varied across tasks: the *no memory* version consistently ran the most generations, indicating the fastest execution speed. The original version performed at an intermediate speed, while both the *reset memory for each agent* and *reset memory for each episode* versions were the slowest, running significantly fewer generations across all tasks due to the blocking of parallelism as discussed in Section 2.5. The blocking mechanism is further illustrated in Algorithm 1. The running time explains the original version's superior results over the same amount of time as indicated in Figure 2.4. However, since this version runs more generations in the same amount of time as *reset for each episode*, it achieves a better score. Still, based on Figure 2.5, it would perform worse if it operated at the same speed as the *reset memory for each episode* case.

These results support our hypothesis that, for tasks without long term state dependencies, resetting memory before each episode can reduce the effect of negative agents and improve results over the same number of generations. As expected, the *no memory* version cannot solve these partially observable tasks.

After training, we reloaded and tested the champion of the last common generations for each case across all seeds. We applied the Mann-Whitney U test to compare each case with the *reset memory for each episode* case, confirming the results in Figure 2.5 with p-values less than 0.05. In Acrobot, the *reset memory for each agent* and *reset memory for each episode* versions showed no significant differences due to similar scores. However, both versions showed significant differences (p-value$<$0.05)

compared to the original and no-memory versions. Readers interested in further details about TPG and visualizations of evolved graphs of teams are referred to [6], [22], [15].

2.6 Conclusion

This study explored the effectiveness of different memory management strategies in enhancing the performance of Tangled Program Graphs in partially observable Reinforcement Learning environments. We experimented with TPG's original shared indexed memory formulation, resetting memory for each agent, resetting memory for each episode, and a no-memory condition across three benchmark tasks: Acrobot, Pendulum, and CartPole.

The results show that resetting memory for each episode improves the performance of TPG agents across all tasks. This strategy led to the highest average scores after the initial few generations and reduced the complexity of decision-making processes. In contrast, the no-memory version, although capable of running more generations, struggled to solve the partially observable tasks effectively, exhibiting the highest complexity and lowest performance.

Interestingly, while the *reset memory for each agent* and *reset memory for each episode* strategies showed similar performance, both were significantly better than the original and no-memory versions in terms of robustness and reliability, demonstrating consistency of the agents' performance across different runs with a tighter distribution of scores over the repeats (Figure 2.5). In contrast, the *reset memory for each generation* case failed to perform better than the original version in CartPole only. The Mann-Whitney U test confirmed these findings, with p-values less than

0.05, indicating significant differences.

These findings suggest that shared memory and "culture" can have a negative impact on the performance of TPG agents in partially observable tasks with no long term temporal dependencies. Resetting memory before each episode can mitigate these negative effects, improving agent performance and reducing decision-making complexity. However, the primary drawback of the memory reset strategies is the increased runtime due to the blocking of parallelism. Implementing a dedicated memory for each agent could potentially mitigate this issue, allowing parallel execution without interference and maintaining computational efficiency.

Overall, effective memory management strategies are crucial in reinforcement learning tasks. By carefully selecting and optimizing memory resetting strategies, significant improvements can be achieved in the efficiency and effectiveness of TPG in challenging control environments.

2.7 Future Work

Future work will scale these experiments to more complex environments, such as Memory Gym [19], in order to validate the methods' robustness and explore their adaptability to tasks with long and short time dependencies. The current memory strategies help agents quickly build mental models without directly sharing information. However, this may not be suitable in complex tasks where global memory is beneficial (e.g. [21]). For such cases, we envision a dynamic method, such as resetting memory based on real-time performance metrics (e.g., wiping memory if median score drops below that of the previous generation), could provide a more adaptive approach. Additionally, investigating other probabilistic memory functions

and their combinations could provide further insights into optimizing agent's memory use. For example, rather than manually resetting memory, it might be possible to *evolve* customized memory management rules for each agent which automatically minimize negative effects on shared memory. Finally, integrating advanced parallelization techniques could mitigate the runtime overhead caused by memory resets, improving their practicality in real-world applications. Since this paper incurred significant wall clock run time, faster TPG frameworks, such as those from [6], will be considered for use in future work.

Overall, studying the long-term evolutionary impacts of different memory strategies could provide deeper insights into the development of more sophisticated and adaptive agents in partially observable environments.

Chapter 3

Genetic Encoding and Shared Knowledge in Reinforcement Learning with Structured Memory

3.1 Abstract

Memory is essential for agents to perform well in partially observable environments, where current input alone is insufficient for decision-making. We investigate this challenge using TPGs, an evolutionary RL framework in which agents are composed of interconnected teams of programs organized into decision-making structures. We introduce a team-specific shared memory mechanism that allows programs within the same team to exchange information during an RL episode, improving coordination without interference from less-closely related agents. We also initialize each program's register memory and team-specific memories with evolved constant values. These constants are evolved through the training process, providing useful starting

points that improve learning and decision-making. We evaluate these strategies on MuJoCo continuous control tasks with partial observability. Our results show that the team-specific shared memory configuration achieves the highest fitness scores across tasks, and that evolved constants improve performance when memory is not retained between timesteps. These findings highlight the importance of learned memory structures and genetic encoding in supporting adaptive behaviour in evolutionary RL systems.

3.2 Introduction

Decision-making in partially observable environments is a fundamental challenge in RL [10]. Many real-world applications, such as robot control, autonomous navigation, and multi-agent coordination, require agents to operate with incomplete information, making it necessary to design models that can retain and share relevant knowledge over time.

TPGs are an evolutionary RL framework based on Genetic Programming (GP). They are composed of graphs of teams, where each team consists of multiple programs acting as decision-making units. This hierarchical structure allows TPGs to evolve solutions through automatic problem decomposition, enabling complex behaviours to emerge from simple components.[12]. While TPGs have demonstrated strong performance in decision-making tasks, their initial implementation lacked an explicit memory mechanism, limiting their effectiveness in environments that require non-sequential dependencies. Prior work introduced a global shared memory, where all agents in the population access a single memory space without any resetting logic [21]. This approach aligns with Spector's [24] "Culture" concept, where individuals

influence one another, facilitating knowledge exchange across the population. However, such shared memory structures introduce the risk of contamination, as poorly performing agents can degrade the quality of stored information. As shown by [5], global shared memory can lead to inefficient information retention and increased complexity. In reinforcement learning, each new interaction sequence with the environment is called an episode. That study also demonstrates that incorporating shared memory into TPGs, when combined with resetting the memory at the beginning of each episode, can lead to improved performance. These findings highlight the need for structured and localized memory management within TPGs.

To address these challenges, we introduce team-specific shared memory, where each TPGs team maintains a dedicated memory space accessible only to its members. Unlike global memory, which allows the entire population to access a single memory space, our approach functions as a smaller society, preserving the benefits of "Culture" while limiting the influence of poorly performing agents. This design prevents interference across teams and allows for parallel execution, as suggested in [5]. Additionally, our method features a scaled normal distribution-based selection mechanism for memory access, optimizing the balance between short-term and long-term memory use. Furthermore, we used TPGs by incorporating three distinct memory structure and associated operations: scalar, vector, and matrix [6], allowing agents to handle richer data representations and solve complex control tasks more effectively. We also incorporate evolved constants in registers and the shared memory, ensuring that agents do not start with zero-initialized values, which can limit early-stage adaptability. Evolved constants allows agents to inherit structured computational biases that improve decision-making. Unlike zero-initialized registers, evolved constants

provide prior knowledge, ensuring agents and teams retain relevant information across generations. This mirrors biological evolution, where evolved constants in registers act as individual genetic encoding [11], while their use in shared memory reflects group-level knowledge retention.

To evaluate our method, we conduct experiments on MuJoCo continuous control tasks, including Inverted Double Pendulum, Hopper, and Half Cheetah. These environments are more complex RL tasks than in prior work, with large, continuous observation and action spaces. To better reflect real-life uncertainty, we introduce a stochastic masking mechanism that mimics natural fluctuations in perception, rather than relying on static observation removal. In particular, MuJoCo environments are used under partial observability conditions by stochastically masking the agent's state observation at each timestep. The effect of inherited knowledge in the form of evolved constants, and stateful registers that retain data throughout an episode is studied. We compare four different configurations: *Stateless Registers - No Constants (SL-NC)*, *Stateless Registers (SL)*, *Stateful Registers (SR)*, and *Stateful Registers with Shared Memory (SR-SM)*.

Our experimental results demonstrate that team-specific shared memory enhances agent performance in partially observable environments. Among the tested configurations, *SR-SM* consistently achieved the highest fitness scores, confirming the benefits of structured memory retention. Additionally, our findings highlight the importance of evolved constants, as agents equipped with inherited knowledge without any memory (*SL*) outperform those with zero-initialized registers (*SL-NC*). However, memory remains essential for handling recurrent dependencies, as *SR* consistently outperforms *SL*.

3.3 Related Work

RL and evolutionary algorithms have been widely explored for decision-making in complex environments. This section reviews key advancements in both, with a focus on methods for partially observable environments.

3.3.1 Neural and Policy-Based RL

Standard RL approaches assume fully observable Markov Decision Processes (MDPs), making them less effective in partially observable Markov decision processes (POMDPs), where only partial state information is available [9]. Several techniques have been proposed to address this limitation.

Recurrent Neural Networks (RNNs), including LSTMs and GRUs, store past information but are computationally expensive [9]. Guided Soft Actor-Critic (Guided SAC) trains an auxiliary agent under full observability to assist the main agent in learning optimal policies [8], but this approach assumes access to a fully observable version of the environment, which may not be available in many real-world or biologically inspired settings. In [31], they introduce Partially Observable Guided RL (PO-GRL), which gradually transitions from full to partial observability, improving policy learning efficiency.

These methods often rely on additional supervision, such as full state access during training, or specialized architectural components like recurrent layers or memory networks. This trend is common in deep RL approaches to partial observability. In contrast, our approach integrates a shared memory mechanism directly into the TPGs structure, reducing state uncertainty without requiring external supervision or additional neural memory structures directly supporting temporal memory such as

recurrent connections.

3.3.2 Evolutionary Algorithms in Partially-Observable Decision-Making

TPGs have been widely studied as an alternative GP framework for evolving decision-making agents [12]. They have demonstrated success in solving RL tasks by evolving graph-based policies in GP rather than using traditional neural networks. However, standard TPGs implementations lack an explicit memory mechanism, which limits their applicability to partially observable RL tasks.

[28] demonstrated that memory is a valuable addition to GP, showing that adding memory mechanisms enhances functionality by allowing programs to retain and use past information. His approach introduced indexed memory in GP, where programs interact with a fixed-size memory array using explicit "read" and "write" instructions. This setup enabled agents to build and update internal models of the environment, supporting more complex, context-aware behaviour without relying on external supervision. In TPGs, memory is typically managed through registers that store intermediate values during decision-making. Expanding this capability, [6] introduced a memory model that extends beyond scalar registers to include vector and matrix operations. In this context, register memory refers to internal program memory—local variables that store and manipulate temporary values within a single program. In contrast, indexed memory (such as shared memory) functions as an external storage space accessible by multiple programs, allowing information to persist and be exchanged across decision steps. This enhancement improved multitask learning and performance in partially observable environments, leading to higher scores and

demonstrating the significant impact of structured memory on TPGs optimization.

In [21], it is observed that agents equipped with external memory are capable of purposefully navigating their environment, whereas those lacking memory are confined to more reactive behaviours such as "flight or fight" responses. This study introduced indexed memory in TPGs, where a single global shared memory was used to store temporary values during execution, meaning that all agents in the population had access to and could modify the same memory space. [5] follows up on this work, finding that a globally shared memory can negatively impact TPGs agents in partially observable control tasks that do not rely on long-term temporal dependencies. In these scenarios, resetting the memory more frequently can improve performance while keeping complexity low and enhancing computational efficiency. In [22], memory enables agents to develop internal state representations, improving decision-making in partial observability. *Context* programs write to memory while *action* programs read from it, highlighting the impact of structured memory in TPGs. These studies demonstrated the value of shared memory but also exposed issues with global access, such as interference between unrelated agents. This inspired our use of team-specific shared memory, limiting access within teams to support more stable and focused information sharing.

To evaluate our proposed extensions, we use MuJoco, a standard benchmark for memory-enhanced RL and evolutionary algorithms in continuous control tasks [30]. Many recent studies [8, 31, 32, 4] have tested structured memory mechanisms in RL models using MuJoCo environments such as Inverted Double Pendulum, Hopper, Half-Cheetah, and Ant. These tasks require precise control, balance, and long-horizon learning, making them ideal for evaluating how memory influences decision-making in

RL. As memory management strategies continue to evolve in TPGs, MuJoCo provides a well-suited platform to assess their effectiveness in complex control scenarios with large continuous observation and action space.

3.4 Methodology

This section outlines the evolutionary framework used in our approach, detailing how agents process information, retain memory, and evolve over generations. We describe the register structure, team-specific shared memory, and the role of evolved constants in enhancing performance.

3.4.1 Tangled Program Graphs (TPGs)

TPGs are an evolutionary learning framework that develop graph-based policies from teams of programs for decision-making tasks [12]. These programs are Linear Genetic Program (LGP) [2] with registers, as illustrated in Figure 3.1. LGPs are also called register machines. Registers function as the internal memory of programs. The data stored in these registers are transformed through a sequence of operations executed by register machines. An example of how LGPs interact with registers is provided in Algorithm 2. The first register, known as the bid value, represents the program's internal estimate of how favourable it would be to execute its associated action, given the current observation and memory state. Higher bid values indicate stronger preferences for execution, influencing which program is selected within a team. The second register stores the selected action.

In TPGs, each agent is represented by teams of programs (Figure 3.2). Each team

Algorithm 2: Example program. Each program consists of scalar (`s`), vector (`v`), and matrix (`m`) memory registers, each with 8 instances. Programs belong to a team and have access to its team specific shared memory, which includes three separate memory types: scalar (`sm`), vector (`vm`), and matrix (`mm`), each with 8 instances. Shared memory operations involve reading and writing values between individual program registers and team memory. For memory write operations (Lines 5 and 7), a memory index is selected probabilistically using a normal distribution centred around the middle of the available memory range. For memory read operations (Lines 8 and 11), values are retrieved from shared memory into program registers.

```
1  v0 = roll(obs(t), -o_i)[:m_w]          ▷ Copy observation to vector memory
2  vi = roll(obs(t), -o_i)[:m_w*m_w]      ▷ Copy observation to temporary vector vi
3  m0 = vi.reshape(m_w,m_w)               ▷ Copy observation to matrix memory
4  v4 = s0*v1                             ▷ Program execution begins
5  scalar_memory_write(s5)                ▷ Write scalar register to shared memory
6  s2 = mean(v3)
7  vector_memory_write(v4)                ▷ Write vector register to shared memory
8  s1 = sm4[3]                            ▷ Scalar memory read
9  s1 = sin(s0)
10 s0 = s0 / v0[3]                        ▷ Observing a value from v0
11 m3 = mm5[1]                            ▷ Matrix memory read
12 s3 = norm(m3)
13 if (s0 < v4[2]) :  s0 = -s3
14 return s0, v1                          ▷ bid, continuous action
```

consists of a group of programs, and each program typically learns to act within a specialized part of the RL environment. Initially, all teams are root nodes pointing to different number of programs, which means each agent of the first generation has just one team. To select an action, all programs are executed, and the team selects the action of the program with the highest bid value. As shown in Figure 3.2, through evolution, program's action can reference other teams within the population, promoting problem decomposition by breaking down complex tasks into smaller, more manageable components. Algorithm 3 refines teams through selection, mutation, and crossover in a structured evolutionary process. The modular, hierarchical structure

of TPGs, combined with memory mechanisms, helps it adapt to partially observable environments by retaining and using past information for better decision-making. As teams evolve, they develop strategies that improve decisions by using memories to retain relevant information. This enables TPGs to be well-suited for tasks such as first-person visual RL, where incomplete world view implies that agents must infer missing information and adapt their strategies accordingly [16].

Algorithm 3: Evolutionary process in TPGs, where T is the population of teams and t_i is an individual team. Each team is evaluated over multiple episodes by executing its programs and selecting actions (Line 6), based on the policy described in [6]. Fitness is averaged across episodes and used to guide selection, mutation (including evolved constants), and crossover.

```
1  Procedure EvolvePopulation(T)                          ▷ T is the set of teams
2      foreach t_i ∈ T do
3          foreach e_j ∈ Episodes do
4              t_i ← LoadeEvolvedConstants(t_i);
5              foreach f ∈ Frames(e_j) do
6                  A_ij ← SelectAction(t_i)     ▷ Run programs and return atomic action
7                  Reward_j += Evaluate(A_ij);
8              F_j += Reward_j;
9          F_i ← F_j / NumberOfEpisodes;
10     if TerminationConditionMet() then
11         return BestTeam(T);
12     T' ← SelectTopTeams(T, F)                                  ▷ Select best teams
13     T'' ← EvolutionOperations(T)                          ▷ Mutation and Crossover
14     T ← ReplaceWeakTeams(T', T'')                  ▷ Combining team population
15     return EvolvePopulation(T);
```

A recent study on TPGs [6] expanded simple scalar register machines to include vector and matrix memory, along with corresponding operations, as shown in Figure 3.1. By incorporating vector and matrix memory into scalar-based TPGs programs, agents gain richer data representations, enhanced decision-making capabilities, and greater adaptability in dynamic environments. Based on the action space, the action

value is either `s1`, for environments like Control task problems, or `v1` to support multi-continuous action environments, like MuJoCo (shown in Figure 3.1 and Algorithm 2). Since this advancement makes TPGs more suitable for solving complex RL problems, it has been used in this study. Algorithm 2 provides an example of how these registers are utilized in this work. Values stored in these registers can either be directly copied from observations (Lines 1, 3 and 10) or computed by applying various operations on existing register values. Further details on the register memory sizes in this work are provided in Table 3.2.

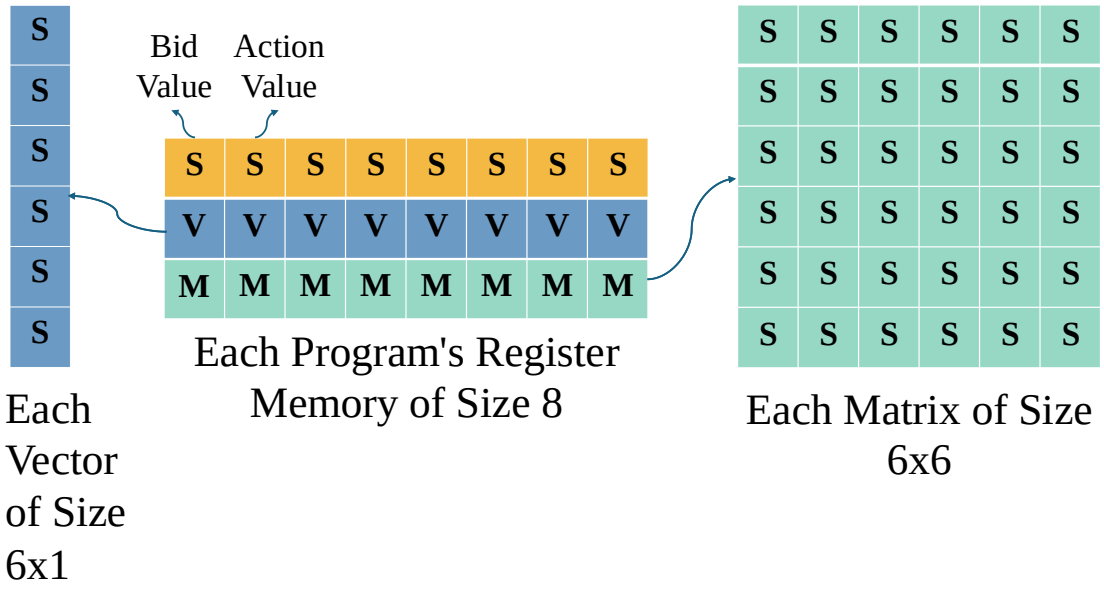


Figure 3.1: Program registers in TPG. This is the register memory structure for an environment with an observation size of 6, which determines the size of each vector and matrix. Each program contains scalar, vector, and matrix registers, where each element stores continuous numerical values. The first element of the scalar register stores the bid value. If the program wins the bid at a given timestep, the action value it returns is either the second element of the scalar register or a value from the vector register, depending on the action space.

3.4.2 Team-Specific Shared Memory TPGs

To overcome global shared memory limitations, we propose a team-specific shared memory model, where each team maintains its own memory space, and all programs of that team have access to that shared memory as indicated in Figure 3.2. In terms of "Culture", we maintain a smaller society, as shown in [5], which has been found to help agents develop their own memory more effectively. This design prevents inter-team interference while enhancing intra-team collaboration, allowing agents to retain and reuse critical information over time.

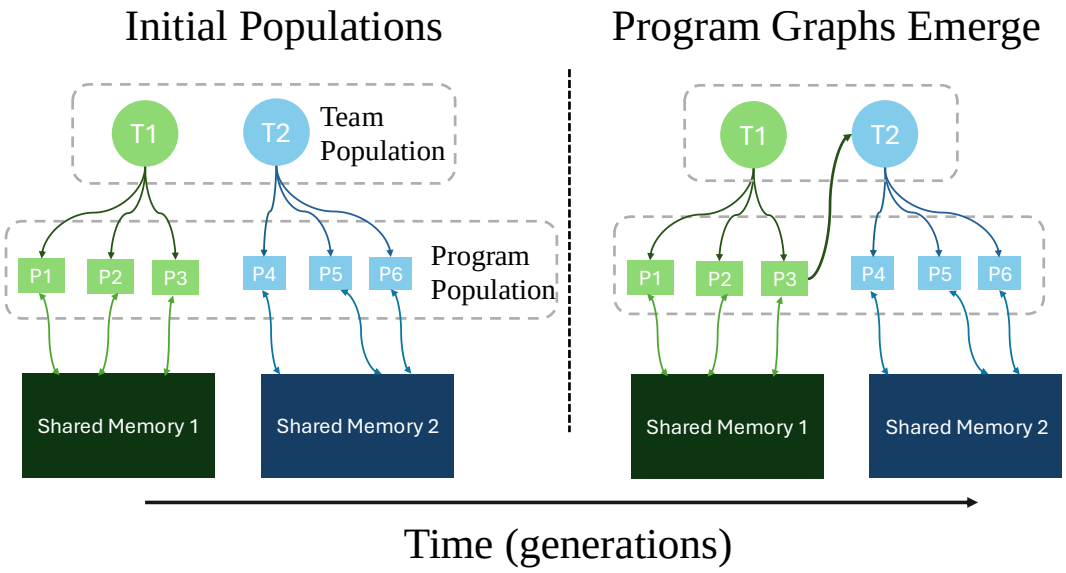


Figure 3.2: Team-Specific Shared Memory TPGs. The figure illustrates the evolution of TPGs with team-specific shared memory. Initially (left), independent teams (T1, T2) contain programs (P1–P6) with separate shared memories. Over generations (right), teams evolve structured connections, forming a hierarchical program graph.

Using the TPGs version from [6], each program operates with scalar, vector, and matrix registers. Accordingly, shared memory is divided into three sections, each

storing data specific to the accessed register type. This shared memory has a size of 8, which means 8 times the registers shown in Figure 3.1. For the "write" instruction (Lines 8 and 11 in Algorithm 2), data is read from the program's registers and written into the corresponding shared memory section. If a scalar register is selected, the value is stored in the scalar memory section; if a vector register is chosen, it is written into the vector memory; and if a matrix register is used, the data is stored in the matrix memory section. Similarly, for the "read" instruction (Lines 8 and 11 in Algorithm 2), data is retrieved from the shared memory and written back into the respective program register. The difference between "write" and "read" is that, for simplicity, the "read" instruction retrieves only a single scalar from the matrix or scalar memory, while the "write" instruction updates all scalars in a selected memory cell. This shared memory with scalar, vector, and matrix parts ensures that memory access remains structured and consistent, enabling efficient data exchange between individual programs and the shared memory space. One example of executing a program with "read" and "write" instruction is provided in Algorithm 2.

In [21], writing is probabilistic, where with a certain probability, register values are written to the columns of the external global memory. In this approach, whenever a register is written to shared memory, the entire memory is updated based on a probabilistic rule. This means that a single write operation requires traversing the entire shared memory structure, deciding for each column, based on a probability, whether it should be updated or not. To ensure efficient memory access for writing, we employ a normal distribution-based selection mechanism, where memory indices are probabilistically sampled (Figure 3.3).

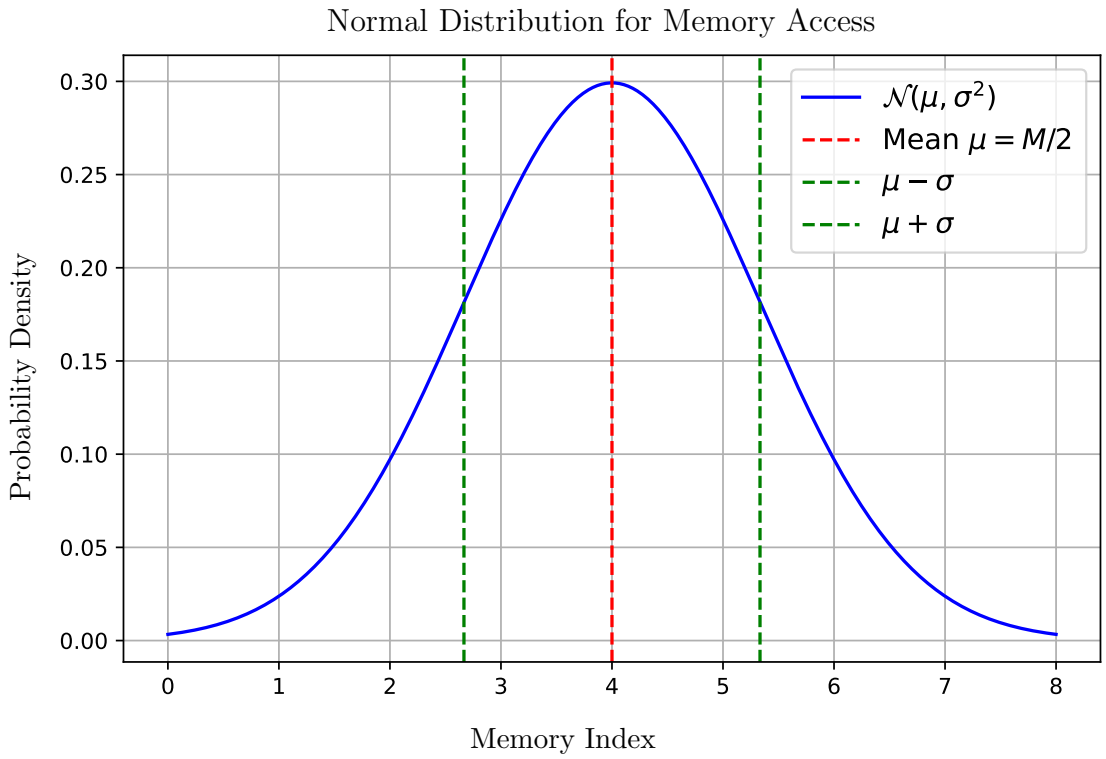


Figure 3.3: Normal distribution used for memory access in team-specific shared memory with $M = 8$. The memory index is sampled from $\mathcal{N}(\mu, \sigma^2)$, with $\mu = M/2$ (red dashed line) and $\sigma = M/6$. The green dashed lines show one standard deviation from the mean, illustrating memory access probability spread.

The normal (Gaussian) distribution is a continuous probability distribution defined by its mean μ and standard deviation σ, with the probability density function:

$$f(i) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(i-\mu)^2}{2\sigma^2}\right), \tag{3.4.1}$$

where i represents a random variable, μ is the mean (center of the distribution), and σ determines the spread. A higher σ results in a wider distribution, while a lower σ concentrates probability values closer to μ.

Since the sampled memory index (i) is a continuous real number, it must be converted into a valid integer memory index to ensure it remains an integer ($i \in \mathbb{Z}$).

This is achieved using the floor function and a clamping operation. As indicated in Figure 3.3, by setting $\mu = \frac{M}{2}$ and $\sigma = \frac{M}{6}$, where $M = 8$, the memory index falls within the range $i \in (0, 8)$. The final chosen memory index is $\lfloor i \rfloor$.

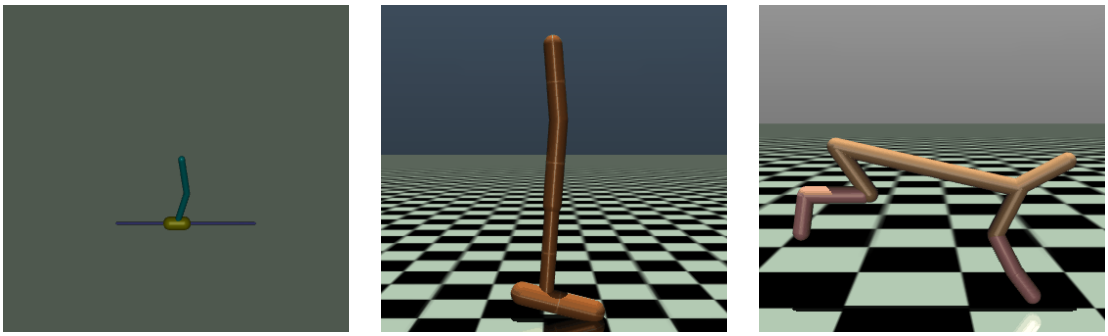
3.4.3 Evolved Constants

Instead of initializing registers and shared memory with zero values, we use evolved constants, which are inherited across generations and refined through mutation. In the first generation, evolved constants are randomly initialized and then evolved thereafter. Following the approach in [20], a real-valued constant is mutated by multiplying it with a uniform random number in the range $[0.5, 2.0]$, with the mutation probability defined in Table 3.2. Additionally, the sign of the constant is flipped with a 10% probability (Algorithm 3, Line 13).

In terms of registers, this provides a prior computational bias, enabling agents to perform meaningful operations at the start of an episode or frame. This mirrors biological heuristics [27], where species inherit evolutionarily advantageous behaviours without requiring immediate learning. For instance, if registers are initialized to zero, adding, subtracting, or multiplying would be meaningless, resulting in less useful information being learned. However, evolved constants address this issue by ensuring that agents start with computational values where structure can emerge through evolution. Regarding shared memory, having evolved constants ensures that shared memory contains evolutionarily useful knowledge, allowing individuals to inherit socially transmitted information. By incorporating these values, TPGs agents are better equipped to handle information flow and execute more effective policies throughout an episode or even a frame.

3.5 RL Tasks

For evaluating the effectiveness of the structured team-specific memory, we selected MuJoCo environments as our benchmarking tasks. MuJoCo is widely used in continuous control tasks and provides highly realistic physics simulations, making it an ideal testbed for assessing memory mechanisms in partially observable environments. By selecting MuJoCo as our test environment, we ensure that our method is evaluated under realistic, high-dimensional, and computationally challenging conditions, making the results more applicable to real-world RL problems. In this work, we studied Inverted Double Pendulum, Hopper, and Half Cheetah (See Figure 3.4). The environment descriptions provided in this section are based on the Gymnasium documentation `https://gymnasium.farama.org/`.



(a) Inverted Double Pendulum Observation: 11, Action: 1

(b) Hopper Observation: 11, Action: 3

(c) Half Cheetah Observation: 17, Action: 6

Figure 3.4: Problem environments used in this work. A higher observation and/or action space means the task is more complex.

3.5.1 Inverted Double Pendulum

In this environment, a cart moves along a straight track with a pole attached to it and a second pole connected to the top of the first (see Figure 3.4a). The second pole, having a free end, makes balancing more challenging. The agent applies continuous forces to move the cart left or right, aiming to keep the second pole balanced while stabilizing the system.

The observation space includes the *cart's position and velocity*, as well as the *angles and angular velocities* of both pendulums. The action space consists of a continuous force in the range of $[-1, 1]$ applied to the cart, allowing precise control over its movement. Each episode has a default maximum duration of 1000 timesteps. The system reaches a height of 1.2 meters when all components are perfectly aligned in a vertical stack. If the height falls below 1 meter, the episode terminates.

The reward function in this environment is defined as:

$$\text{reward} = \text{alive\_bonus} - \text{distance\_penalty} - \text{velocity\_penalty} \tag{3.5.1}$$

alive_bonus is a fixed reward (healthy_reward = 10) given at each timestep while the pendulum remains upright. The distance_penalty measures how far the tip of the second pole deviates from its ideal position and is computed as:

$$0.01(x^2_{\text{pole2-tip}} + (y_{\text{pole2-tip}} - 2)^2) \tag{3.5.2}$$

where $x_{\text{pole2-tip}}, y_{\text{pole2-tip}}$ represent the tip's xy-coordinates. This encourages the

agent to minimize unnecessary displacement. Additionally, the velocity_penalty discourages excessive motion by applying a negative reward based on the angular velocities of the two joints, given by:

$$10^{-3}\omega_1 + 5 \times 10^{-3}\omega_2 \tag{3.5.3}$$

where ω_1, ω_2 are the angular velocities of the first and second hinges, respectively. The agent maximizes rewards by maintaining stability and reducing unnecessary movement for smoother, controlled balancing.

3.5.2 Hopper

The Hopper is a two-dimensional, single-legged robot with four main body segments: a torso, thigh, leg, and foot (See Figure 3.4b. The agent controls torques at three joints to produce coordinated hopping, aiming to move forward efficiently while maintaining balance. Successful locomotion depends on precise joint coordination and stability, making this a challenging continuous control task.

The observation space includes the torso's position, velocity, height, movement along the x-axis, angle, and angular velocity, as well as the angles and angular velocities of the thigh, leg, and foot joints. The action space consists of three continuous torques applied to the hip, knee, and ankle joints, enabling dynamic posture adjustments to maintain forward momentum. Each episode has a maximum duration of 1000 timesteps but may terminate earlier if the Hopper becomes unhealthy. Termination occurs if any state variable exceeds the limits set by the `healthy_state_range`, including excessive joint angles or velocities. The episode also ends if the Hopper falls, indicated by its height dropping below 0.7 meters, or if the torso angle deviates

beyond the range $[-0.2, 0.2]$ radians.

The reward function in the Hopper environment is designed to encourage forward movement while penalizing excessive control efforts. The total reward is calculated as:

$$\text{reward} = \text{healthy\_reward} + \text{forward\_reward} - \text{ctrl\_cost} \tag{3.5.4}$$

The healthy reward is a fixed value given at each timestep while the Hopper remains in a valid state, with a default value of 1. The forward reward incentivizes movement in the positive x-direction and is computed as

$$w_{\text{forward}} \times \frac{dx}{dt} \tag{3.5.5}$$

where dx represents the displacement of the torso, and dt is the time step between actions. The value of dt is determined by the frame skip parameter (default: 4) and the frametime (0.002 s), resulting in $dt = 4 \times 0.002 = 0.008$.

The weighting factor w_{forward} (default: 1) scales the contribution of forward movement to the total reward. The control cost penalty discourages excessive action magnitudes by applying a negative reward based on the squared norm of the action vector:

$$w_{\text{control}} \times \|action\|_2^2 \tag{3.5.6}$$

where w_{control} is the control cost weight, set to a default value of 10^{-3}. Together, these reward components encourage the agent to develop efficient, stable, and energy-conscious locomotion.

3.5.3 Half Cheetah

The Half Cheetah is a two-dimensional robotic model with nine body segments connected by eight joints, including two paws (see Figure 3.4c). The goal is to control torque at six joints to maximize forward speed, earning a positive reward for forward movement and a negative reward for moving backward. The torso and head remain fixed, while torque is applied to the front and back thigh joints (attached to the torso), the shin joints (connecting to the thighs), and the foot joints (connecting to the shins).

The observation space includes the front tip's height, position, and velocity along the x- and z-axes, as well as its angle and angular velocity. It also contains the angles and angular velocities of the back thigh, back shin, back foot, front thigh, front shin, and front foot joints. The x-coordinate of the front tip is excluded. The action space consists of six continuous torque values applied to the hinge joints. These torques control the movement of the back thigh, back shin, back foot, front thigh, front shin, and front foot. Each torque value ranges from $[-1, 1]$, allowing the agent to adjust joint forces for efficient locomotion. This environment does not have termination conditions; episodes continue until they reach the maximum length of 1000 timesteps.

The total reward is defined as:

$$\text{reward} = \text{forward\_reward} - \text{ctrl\_cost} \tag{3.5.7}$$

The forward reward encourages movement in the positive x-direction and is computed as shown in Equation 3.5.5. In this environment, the value of dt is determined by the frame skip parameter (default: 5) and the frametime (0.01 s), resulting in $dt = 5 \times 0.01 = 0.05$.

The control cost applies a penalty for excessive action magnitudes, given by the Equation 3.5.6 where w_{control} (default: 0.1) is the control cost weight. The control cost discourages overly aggressive actions, promoting more stable and efficient movement.

3.5.4 Partial Observability

To introduce partial observability, we implement a stochastic masking mechanism. At each step, the method randomly determines whether the agent receives full state information or a completely masked (blind) observation, controlled by sampling from a uniform distribution over $[0, 3)$. If the random number is less than 1, the observation vector is set to zero. Thus, the agent receives full state information with probability $\frac{1}{3}$ and no state information with probability $\frac{2}{3}$. As a result, the agent can no longer rely solely on the current observation but must instead infer missing information from past interactions to make optimal decisions. This stochastic masking approach more closely reflects real-world perceptual uncertainty, where agents must operate under intermittent or unpredictable access to sensory information.

3.6 Experiments

This section presents the experimental setup used to evaluate different memory mechanisms in TPGs. We compare four approaches: *Stateless Registers - No Constants (SL-NC)*, *Stateless Registers (SL)*, *Stateful Registers (SR)*, and *Stateful Registers with Shared Memory (SR-SM)* agents. In all tests, except for *SL-NC*, evolved constants are used, as described in Section 3.4.3. Table 3.1 summarizes the specific memory features active in each configuration to clarify what is being tested.

Variants	Evolved Constants	Register Memory	Shared Memory
SL-NC	-	-	-
SL	✓	-	-
SR	✓	✓	-
SR-SM	✓	✓	✓

Table 3.1: Active memory components per variant.

The goal is to analyze the impact of structured memory on decision-making performance in partially observable RL tasks. The detailed parameters and configurations utilized in this work are listed in Table 3.2. For Inverted Double Pendulum, the population size is 1000 root teams with 300 new teams per generation, whereas the more complex Hopper and Half Cheetah require 1500 root teams and 1000 new teams per generation for better exploration.

Parameter	Value
Tournament size	3
Evolved constants mutation probability	0.5
Team crossover probability	0.5
Program mutation probability	0.1
Program addition probability	0.075
Program deletion probability	0.1
Action pointer mutation probability	0.1
Atomic action selection probability	0.9
Number of memory registers	8
Memory size	Observation size

Table 3.2: Key experimental parameters used in the evolutionary process, including population dynamics, mutation rates, memory configuration, and action selection probabilities.

3.6.1 Stateless Registers - No Constants (SL-NC)

In this version, register memories are stateless, meaning they do not carry information from one step to the next. At the beginning of each execution, the registers are initialized with zeros. This setup tests whether the task can be solved without any memory or if, due to partial observability, it actually needs more than just the observation at each step.

3.6.2 Stateless Registers (SL)

This approach is similar to the previous one, with the key difference being that at the beginning of each execution, the registers are initialized with evolved constants. Applying operations on nonzero values allows programs to exhibit some degree of adaptation in partially observable environments, preventing them from relying purely on the current observation. However, without retaining information from previous observations, the agent's performance remains limited in tasks requiring long-term dependencies.

This setting tests the ability of TPGs to perform without any form of memory from previous steps in partially observable tasks. It also examines the added value of initializing memory with evolved constants.

3.6.3 Stateful Registers (SR)

In this setup, agents rely exclusively on the registers shown in Figure 3.2 to store temporary values during executions. Register memories are set to evolved constants prior to each evaluation and left to accumulate throughout the episode. There is no mechanism for sharing information across programs. This approach serves as a

baseline memory mechanism to assess the effectiveness of more advanced memory structures.

3.6.4 Stateful Registers with Shared Memory (SR-SM)

This setup extends the previous approach, where each program has its own register memories, as described in Section 3.6.3. The extension introduces team-specific shared memory, as detailed in Section 3.4.2. The shared memory is initialized with evolved constants at the beginning of each episode. This design enables information retention and sharing across programs within each team, facilitating better decision-making.

3.7 Results

In this section, we compare fitness scores for each task using the reward functions defined in Section 3.5. We also measure computational complexity as the number of executed instructions per prediction to assess performance across TPGs memory setups over 10 repeats.

Variants	Inverted Double Pendulum	Hopper	Half Cheetah
SL-NC	1630.16	1619.82	1.58
SL	9349.37	2017.00	608.98
SR	9349.92	2165.23	376.77
SR-SM	**9350.03**	**3078.25**	**725.97**

Table 3.3: Best results on MuJoCo tasks. The best result in each task column is shown in bold.

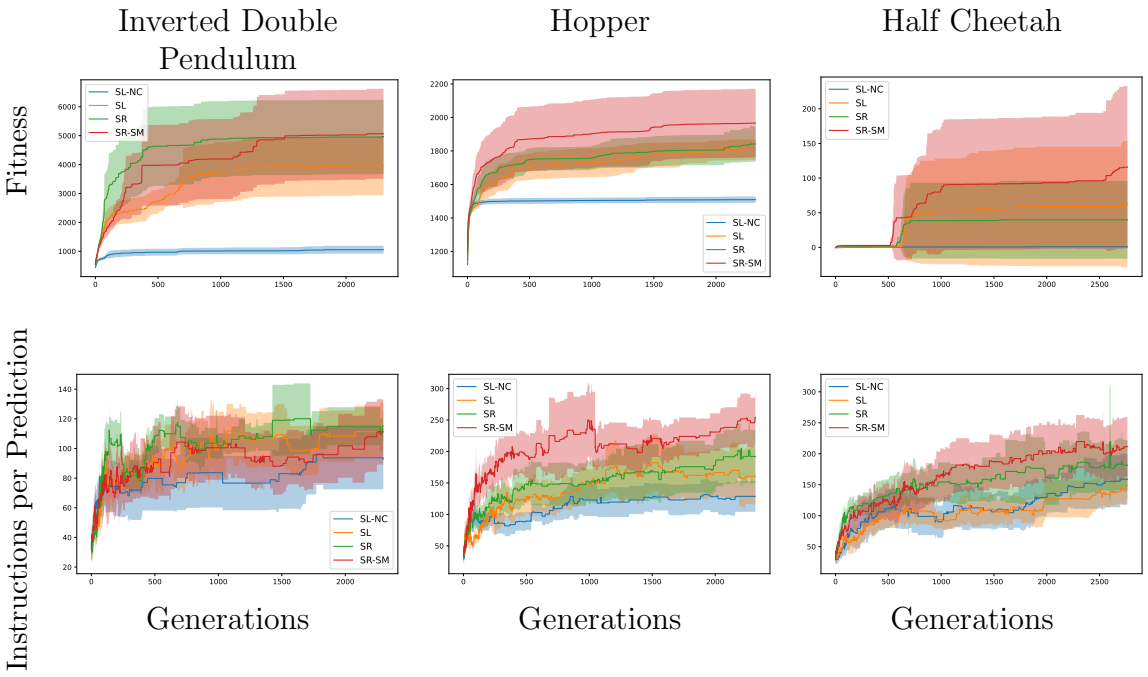


Figure 3.5: Comparison of different memory setups in TPGs across 10 repeats that shows the effectiveness of different memory configurations. **Top row**: Mean fitness (solid line) and standard deviation for the 10 repeats. **Bottom row**: The corresponding instructions used per prediction for the 10 repeats.

Table 3.3 presents the best fitness scores achieved in each environment, demonstrating the effectiveness of using shared memory. Based on this table, *SR-SM* achieves the highest score among all tests. In Inverted Double Pendulum, highest possible score is achieved. Moreover, after visualizing the Hopper with shared memory, it can be seen that it hops. Consequently, in at least one of the 10 runs, *SR-SM* was able to solve the task in Inverted Double Pendulum and Hopper. This suggests that structured memory retention plays a crucial role in maintaining useful information across timesteps, allowing the agent to make more informed decisions in partially observable settings.

Although the best possible score in Hopper is more than 3500, our test can hop,

which becomes evident when visualized. The main challenge we encountered was Half Cheetah, where none of the tested setups were able to approach the best possible score, which is above 10,000, due to the task's high complexity. This is because of the fact that a probability of $\frac{1}{3}$ for receiving an observation makes the task unsolvable in this environment, unlike the other two. Allowing for more generations in the evolutionary process may help agents develop more effective memory strategies, potentially improving performance in this task. Consequently, a higher chance of obtaining observations is required to solve the task with memory while maintaining partial observability.

Figure 3.5 emphasizes the impact of using shared memory, as it achieves the highest mean fitness value among all setups across all 3 environments. Moreover, in this figure *SR-SM* shows the highest increase in complexity over time, indicating complexification [25]. This increase indicates that the agent continues to refine its strategy over time, starting from a simple form and evolving more complex behaviours through interaction with the environment. Inverted Double Pendulum is easier compared to the other two can also be inferred from the low variance between different test results. In this environment, while *SL-NC* fails to solve the task, *SL* achieves a best fitness score that can be considered as solving the task, as shown in Table 3.3. This further reinforces the benefit of evolved constants while also confirming the relatively low difficulty level of this environment.

In terms of the importance of using evolved constants, the difference between *SL* and *SL-NC* in Figure 3.5 and Table 3.3 supports our expectation, as explained in Section 3.6, that having these evolved values instead of 0 has a significant impact on agent performance. Despite the fact that evolved constants improve performance

compared to using zeros, *SL* still cannot achieve better performance than *SR*, highlighting the necessity of register memory. The superiority of *SR* over *SL* is evident from Table 3.3 and Figure 3.5, where, after *SR-SM*, *SR* achieves the highest fitness and highest mean fitness. The only task where *SL* performs better than *SR* is Half Cheetah, but since this task remains unsolved, no meaningful conclusion can be drawn from this result. Moreover, Figure 3.5 illustrates that in all three tasks, including Half Cheetah, the average number of executed instructions per decision across 10 different runs is higher for *SR* than for *SL-NC*, which can be interpreted as *SR* making more attempts to solve the tasks.

The best agent for each task was reloaded for 100 episodes, and the Mann-Whitney U test was used to compare the results with a significance level of $p = 0.05$. Each environment's variants were compared pairwise, and all showed significant differences except for *SR-SM* and *SL* in Half Cheetah and Hopper. In the case of Half Cheetah, this is likely because the task was not solved. For Hopper, the lack of a significant difference may suggest that more generations are needed to produce stable hopping behaviour, which could lead to more reliable performance during the test phase. However, what is most important in this context is that the highest score in Hopper was achieved by *SL*.

Figure 3.6 illustrates how shared memory is used by the best-performing agents in Inverted Double Pendulum and Hopper. Since Half Cheetah was not solved, its memory usage is not included in this analysis. Most notably, all write operations occur exclusively in the matrix memory, with no writing to the scalar or vector sections. This suggests that, through evolution, agents have learned to favour matrix memory as the most effective for coordination. The use of the normal distribution in write

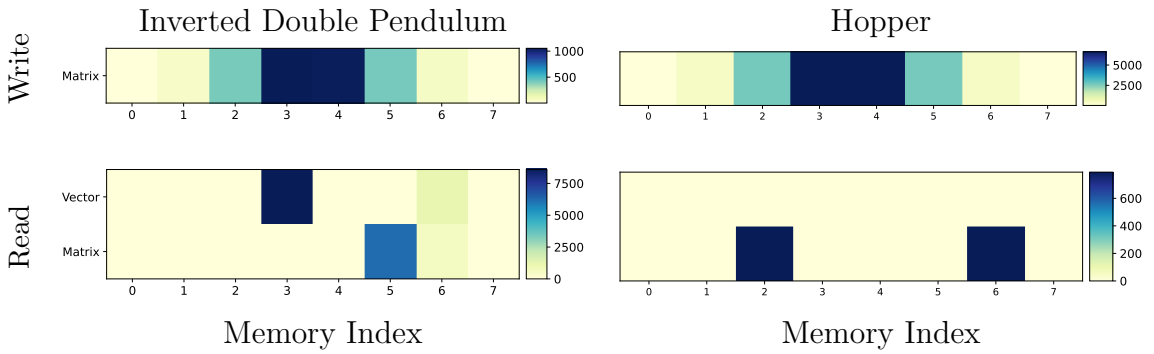


Figure 3.6: Average number of shared memory interactions in the best-performing run from Table 3.3, measured over 100 episodes for both solved tasks. **Top and bottom rows**: Average number of "write" and "read" instructions, respectively.

operations is clearly reflected in the memory access pattern shown in the figure.

For read operations, Inverted Double Pendulum demonstrates shorter-term memory dependencies compared to Hopper, due to more frequent updates in the middle of the memory following the normal distribution. The impact of evolved constants is also observed in the read pattern: in Inverted Double Pendulum, the agent reads from vector memory despite no write operations to that section, meaning it is accessing only the evolved constants.

Figure 3.7 presents the average number of shared memory interactions per program for the best-performing agents in the two solved tasks. These agents consist of a single team and do not form tangled graphs. The figure reveals that some programs interact with shared memory more than others, while some exhibit no interaction at all. This likely reflects the specialized roles that different programs take on within the team, as described in Section 3.4.1, where each program is responsible for handling specific aspects of the environment.

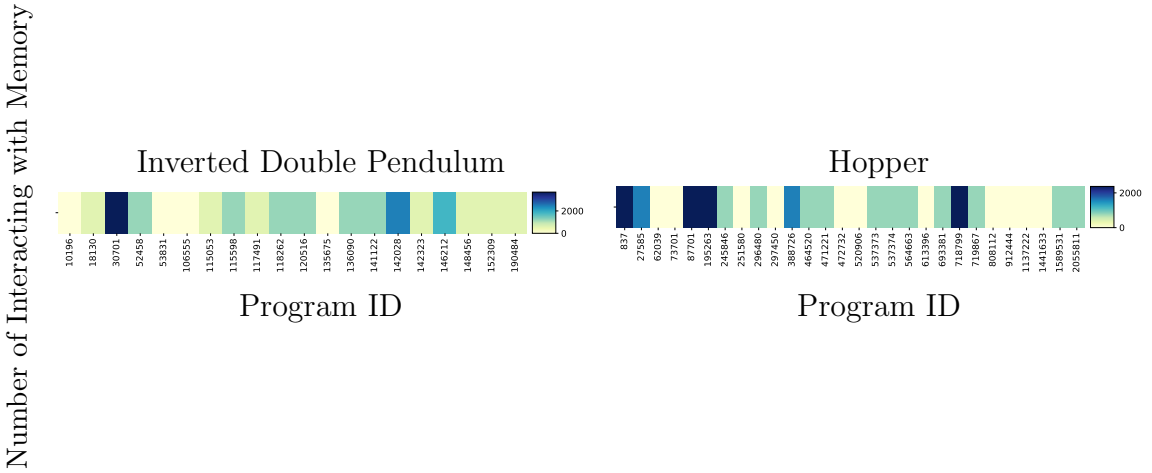


Figure 3.7: Average number of shared memory interactions per program, combining both "write" and "read" instructions. Results are averaged over 100 episodes using the best-performing agent from Table 3.3 for the two solved tasks. Each of these agents consists of a single team, without any tangled graph structure.

3.8 Conclusion and Future Work

In this study, we introduced team-specific shared memory for TPGs to enhance performance in partially observable RL environments. This method enables each team to maintain a separate memory space, improving information retention and, we hypothesize, reducing interference between unrelated programs.

Our experimental results on MuJoCo continuous control tasks demonstrate that structured memory improves agent performance in partially observable environments. Among the tested configurations, *SR-SM* consistently achieved the highest fitness scores across different tasks, confirming the benefits of team-specific shared memory. Compared to agents without this memory mechanism, it facilitates longer-term information retention, enabling more effective strategy refinement and adaptation in complex environments.

Moreover, our findings highlight the importance of evolved constants in improving agent performance. The comparison between *SL* and *SL-NC* confirms that using evolved constants instead of zero-initialized registers enhances decision-making capabilities. However, our results also indicate that while evolved constants improve performance, memory remains essential for solving more challenging tasks. This is evident as *SR* consistently outperforms *SL*, emphasizing the necessity of register memory for handling episodic dependencies.

This study shows that, through evolution, matrix memory is used more frequently than scalar or vector memory. Based on this finding, future work could explore using only matrix memory as the shared memory structure to reduce complexity and compare its performance with the current approach. Additionally, the current implementation of the read operation retrieves only a single scalar value from each section. This could be extended to allow reading an entire vector or matrix from a selected index, similar to how the write operation is performed. It would also be worthwhile to investigate agent performance more closely to see whether some agents consistently perform poorly and become destructive when interacting with global memory. More generally, future work could explore whether there are truly "good" and "bad" agents in the population, and how their presence affects shared memory dynamics. All of these suggestions, as well as the introduced shared memory and evolved constants, could be applied to time series tasks, where identifying and remembering trends may lead to improved predictions.

Chapter 4

Conclusion

Genetic Programming (GP) offers strong interpretability due to its use of symbolic operations, transparent structure, and sparse feature usage. In GP, programs are typically compact and modular, often relying on only a small subset of input features, which makes it easier to trace how decisions are made. Tangled Program Graphs (TPGs) build on this by introducing a hierarchical, graph-based policy structure that further enhances interpretability [22].

Memory management strategies and the exploration of efficient memory structures play a crucial role in optimizing the performance of TPGs agents in RL tasks. In particular, memory becomes essential in partially observable environments, where agents must rely on past observations to make informed decisions. Since memory allows agents to retain and integrate information across time steps, designing effective memory mechanisms is key to enabling them to construct internal models of their environment.

Both studies presented in this thesis focus on enhancing the use of indexed memory within TPGs and demonstrate how strategic memory design can lead to improved

agent performance.

The first study examined different memory management strategies within a globally shared memory setup, where all agents had access to the same memory space. Using benchmark control tasks with short temporal dependencies, Acrobot, Pendulum, and CartPole, it was shown that resetting memory at the start of each RL episode boosts performance and reduces decision-making complexity. These findings highlight the importance of mitigating memory contamination caused by underperforming agents when memory is shared across the entire population.

Building on these insights, the second study introduced a team-specific shared memory model. We hypothesize that by restricting memory access to programs within the same team, this approach preserves intra-team knowledge while preventing negative interference from unrelated agents from other teams. In addition, the integration of evolved constants into both register and memory initialization provided agents with useful priors, enhancing their learning capabilities. Experiments on continuous control tasks from the MuJoCo suite, including Inverted Double Pendulum, Hopper, and Half Cheetah, demonstrated that structured and localized memory, when combined with genetic encoding, results in more adaptive and effective learning strategies.

Together, these contributions elaborate how memory can be purposefully structured and leveraged in evolutionary RL systems. By showing that both the scope and the initialization of memory affect learning outcomes, this work offers valuable guidance for the continued development of interpretable and efficient RL frameworks such as TPGs. These findings also suggest promising directions for future work, particularly in scaling memory-aware TPGs to environments with longer time dependencies and greater complexity.

Bibliography

[1] R. Amaral. Pytpg: Tangled program graphs in python. `https://github.com/Ryan-Amaral/PyTPG/tree/7295f90ececbfc34fdbc1d73e032a9c2407a182c`, 2019.

[2] M. Brameier and W. Banzhaf. *Linear Genetic Programming*. Springer, 2007.

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI Gym. *arXiv*, 1606.01540, 2016.

[4] K. Chen, Z. Gan, S. Leng, and C. Guan. Deep reinforcement learning with parametric episodic memory. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7, 2022. doi: 10.1109/IJCNN55064.2022.9891902.

[5] T. Djavaherpour, A. Naqvi, and S. Kelly. Tangled program graphs with indexed memory in control tasks with short time dependencies. In *Proceedings of the 16th International Joint Conference on Computational Intelligence - Volume 1: ECTA*, pages 296–303. INSTICC, SciTePress, 2024. ISBN 978-989-758-721-4. doi: 10.5220/0013016800003837.

[6] T. Djavaherpour, A. Naqvi, E. Zhuang, and S. Kelly. *Evolving Many-Model Agents with Vector and Matrix Operations in Tangled Program Graphs*, pages

87–105. Springer Nature Singapore, Singapore, 2025. ISBN 978-981-96-0077-9. doi: 10.1007/978-981-96-0077-9_5. URL https://doi.org/10.1007/978-981-96-0077-9_5.

[7] A. E. Eiben and J. E. Smith. *Introduction to evolutionary computing*. Springer, 2015.

[8] M. Haklidir and H. Temeltaş. Guided soft actor critic: A guided deep reinforcement learning approach for partially observable markov decision processes. *IEEE Access*, 9:159672–159683, 2021. doi: 10.1109/ACCESS.2021.3131772.

[9] M. J. Hausknecht and P. Stone. Deep recurrent q-learning for partially observable mdps. *ArXiv*, abs/1507.06527, 2015. URL https://api.semanticscholar.org/CorpusID:8696662.

[10] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1): 99–134, 1998. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(98)00023-X. URL https://www.sciencedirect.com/science/article/pii/S000437029800023X.

[11] Y. Kassahun, M. Edgington, J. H. Metzen, G. Sommer, and F. Kirchner. A common genetic encoding for both direct and indirect encodings of networks. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, page 1029–1036, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936974. doi: 10.1145/1276958.1277162. URL https://doi.org/10.1145/1276958.1277162.

[12] S. Kelly and M. I. Heywood. Emergent tangled graph representations for atari game playing agents. In J. McDermott, M. Castelli, L. Sekanina, E. Haasdijk, and P. García-Sánchez, editors, *Genetic Programming*, pages 64–79, Cham, 2017. Springer International Publishing. ISBN 978-3-319-55696-3.

[13] S. Kelly and M. I. Heywood. Emergent Solutions to High-Dimensional Multitask Reinforcement Learning. *Evolutionary Computation*, 26(3):347–380, 09 2018. ISSN 1063-6560. doi: 10.1162/evco_a_00232. URL https://doi.org/10.1162/evco_a_00232.

[14] S. Kelly, J. Newsted, W. Banzhaf, and C. Gondro. A modular memory framework for time series prediction. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20, page 949–957, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371285. doi: 10.1145/3377930.3390216. URL https://doi.org/10.1145/3377930.3390216.

[15] S. Kelly, R. J. Smith, M. I. Heywood, and W. Banzhaf. Emergent tangled program graphs in partially observable recursive forecasting and vizdoom navigation tasks. *ACM Trans. Evol. Learn. Optim.*, 1(3), aug 2021. doi: 10.1145/3468857. URL https://doi.org/10.1145/3468857.

[16] S. Kelly, R. J. Smith, M. I. Heywood, and W. Banzhaf. Emergent tangled program graphs in partially observable recursive forecasting and vizdoom navigation tasks. *ACM Transactions on Evolutionary Learning and Optimization*, 1(3):1–41, 2021.

[17] J. R. Koza. Genetic programming as a means for programming computers by

natural selection. *Statistics and Computing*, 4(2):87–112, 1994. doi: 10.1007/BF00175355. URL https://doi.org/10.1007/BF00175355.

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015. URL https://api.semanticscholar.org/CorpusID:205242740.

[19] M. Pleines, M. Pallasch, F. Zimmer, and M. Preuss. Memory gym: Partially observable challenges to memory-based agents. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=jHc8dCx6DDr.

[20] E. Real, C. Liang, D. R. So, and Q. V. Le. Automl-zero: evolving machine learning algorithms from scratch. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[21] R. J. Smith and M. I. Heywood. A model of external memory for navigation in partially observable visual reinforcement learning tasks. In *Genetic Programming: 22nd European Conference, EuroGP 2019, Held as Part of EvoStar 2019, Leipzig, Germany, April 24–26, 2019, Proceedings*, page 162–177, Berlin, Heidelberg, 2019. Springer-Verlag. ISBN 978-3-030-16669-4. doi: 10.1007/978-3-030-16670-0_11. URL https://doi.org/10.1007/978-3-030-16670-0_11.

[22] R. J. Smith and M. I. Heywood. Interpreting tangled program graphs under

partially observable dota 2 invoker tasks. *IEEE Transactions on Artificial Intelligence*, 5(4):1511–1524, 2024. doi: 10.1109/TAI.2023.3279057.

[23] L. Spector and S. Luke. Cultural transmission of information in genetic programming. In *Proceedings of the 1st Annual Conference on Genetic Programming*, page 209–214, Cambridge, MA, USA, 1996. MIT Press. ISBN 0262611279.

[24] L. Spector and S. Luke. Culture enhances the evolvability of cognition. In G. Cottrell, editor, *Cognitive Science (CogSci) 1996 Conference Proceedings*, pages 672–677, Mahwah, NJ, USA, 1996. Lawrence Erlbaum Associates. URL `http://www.cs.gmu.edu/~sean/papers/culture-cogsci.pdf`.

[25] K. O. Stanley and R. Miikkulainen. Competitive coevolution through evolutionary complexification. *J. Artif. Int. Res.*, 21(1):63–100, Feb. 2004. ISSN 1076-9757.

[26] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, 2nd edition, 2018.

[27] B. Sznajder, M. W. Sabelis, and M. Egas. How adaptive learning affects evolution: Reviewing theory on the baldwin effect. *Evolutionary Biology*, 39:301 – 310, 2011. URL `https://api.semanticscholar.org/CorpusID:255342397`.

[28] A. Teller. Learning mental models. In *PROCEEDINGS-SPIE THE INTERNATIONAL SOCIETY FOR OPTICAL ENGINEERING*, pages 147–147. Citeseer, 1993.

[29] A. Teller. *The evolution of mental models*, page 199–217. MIT Press, Cambridge, MA, USA, 1994. ISBN 0262111888.

[30] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012. doi: 10.1109/IROS.2012.6386109.

[31] S. Weigand, P. Klink, J. Peters, and J. Pajarinen. Reinforcement learning using guided observability. *ArXiv*, abs/2104.10986, 2021. URL `https://api.semanticscholar.org/CorpusID:233347091`.

[32] M. Zhang, F. Qian, and Q. Liu. Memory sequence length of data sampling impacts the adaptation of meta-reinforcement learning agents, 2024. URL `https://arxiv.org/abs/2406.12359`.