

Replacing REINFORCE with Evolutionary Algorithms in GPTSwarm: A Reproduction and Modification Study

Baran Akkanat Doga Turak
Madalina Gavat

June 2026

Abstract

Multi-agent large language models can improve task performance by optimizing communication topology between agents. This is formalized by representing agents as nodes in a directed acyclic graph and using REINFORCE to learn edge probabilities that suppress adversarial agents, operating on a continuous relaxation of an inherently discrete problem that produces high-variance gradient estimates and may be suboptimal for combinatorial graph search. In this work, we reproduce the adversarial MMLU setting and replace the REINFORCE-based edge optimizer with an evolutionary algorithm that directly evolves binary adjacency matrices, eliminating the need for gradient computation and the discretization step required at inference time. We evaluate five conditions, a single IO agent baseline, a fully connected swarm, a randomly connected swarm, REINFORCE, and our EA, across four swarm configurations (1T1A, 3T3A, 5T5A, 7T7A) using Llama 3.1:8b via Ollama. Both optimizers elevate from the random connectivity baseline and approach the fully connected accuracy ceiling, with REINFORCE and EA achieving 50% and 51% respectively on 3T3A. However, EA has substantially higher token cost, consuming approximately $3.7\times$ more tokens than REINFORCE at 3T3A and up to $10\times$ more at 7T7A, without a noticeable accuracy gain, proving that EA is inefficient across REINFORCE and fully connected graphs.

1 Introduction

Recent work on multi-agent LLM systems has demonstrated that structuring communication between agents can substantially improve task performance. Frameworks such as AutoGen [2] and MetaGPT [3] allow multiple specialized agents to collaborate by exchanging outputs and dividing reasoning steps across roles. DyLAN [4] takes this further by dynamically selecting which agents communicate based on task requirements, achieving 75.1% on MMLU with GPT-3.5, outperforming static single-agent configurations. A shared observation across these systems is that *how* agents are connected matters as much as *what* each agent does. Despite this, the communication topology in most frameworks is either fixed by hand or determined by simple heuristics rather than optimized explicitly.

The authors from [1] address this limitation by treating the communication topology of a multi-agent system as a learnable parameter. In the GPTSwarm framework, agents are represented as nodes in a directed acyclic graph, and edges encode information flow between them. The system supports two levels of optimization: node-level refinement of agent prompts, and edge-level optimization of the communication graph itself. For edge optimization, GPTSwarm uses REINFORCE [5]: each edge is parameterized by a logit, converted to a probability via sigmoid, and sampled using a Bernoulli distribution during each forward pass. Gradient updates are derived from a binary reward signal reflecting task correctness. The paper evaluates this approach on the MMLU benchmark [6] in an adversarial setting where a subset of agents is instructed to provide incorrect answers. REINFORCE learns to suppress adversarial agents by reducing their connection probability to near zero, recovering accuracy that would otherwise be degraded by corrupted inputs.

However, applying REINFORCE to graph topology optimization introduces a structural mismatch: communication graphs are inherently discrete objects, but REINFORCE operates on a continuous relaxation of this discrete problem. The optimizer manipulates edge probabilities rather than edges directly, and the final topology must be recovered by thresholding these probabilities at inference time. Furthermore, REINFORCE is known to produce high-variance gradient estimates [5], and this is amplified in the MMLU setting by small batch sizes and a sparse binary reward signal. These characteristics suggest that REINFORCE may be a suboptimal choice for a problem that is fundamentally combinatorial in its nature.

Evolutionary algorithms [7] offer a natural alternative. They operate directly on discrete representations, require no gradient computation, and explore the search space through a population of candidate solutions rather than a single gradient trajectory. In this work, we reproduce the adversarial MMLU setting from (author?) [1] and replace its REINFORCE-based edge optimizer with an evolutionary algorithm that directly evolves binary adjacency matrices. Our research question is: *Can an evolutionary algorithm optimize GPTSwarm topologies more effectively and efficiently than REINFORCE in an adversarial*

MMLU setting? We evaluate this by comparing five conditions: a direct-answer single-agent baseline, a fully connected swarm, a randomly connected swarm, the original REINFORCE-based GPTSwarm, and our EA-based modification, measuring both accuracy and token consumption across conditions.

2 Methodology

2.1 Graph Definition

Following the GPTSwarm framework, we represent each LLM-based agent as a directed acyclic graph (DAG) and a swarm of agents as a composite graph.

2.1.1 Single Agent Graph

A single agent is represented as a directed acyclic graph $G = (N, E, F, o)$, where:

- N is the set of computational nodes, each representing a fundamental operation. In our case operations are LLM inference or aggregation.
- $E \subseteq N \times N$ is the set of directed edges encoding information flow between nodes
- $F = \{f_n\}_{n \in N}$ is the set of computational routines, where f_n processes node n 's inputs
- $o \in N$ is the output node whose result defines the agent's final answer

Given input x , graph execution proceeds in topological order: node n receives input x and context z_n from its predecessors, applies $f_n(z_n, x)$, and forwards its output to successor nodes. The final output is $\hat{y} = f_o(z_o, x)$.

2.1.2 Composite Graph (Swarm)

A swarm of K agents is represented as a composite graph $G_E = (N', E_E, F', o')$, where:

- $N' = \bigcup_{k=1}^K N_k$ is the union of all nodes across agents
- $E_E = E' \cup E_{\text{inter}}$ combines intra-agent edges $E' = \bigcup_{k=1}^K E_k$ and inter-agent edges $E_{\text{inter}} \subseteq \bigcup_{i \neq j} N_i \times N_j$
- $F' = \bigcup_{k=1}^K F_k$ is the union of all computational routines
- o' is the output node (typically an aggregation node receiving votes from all agents)

The inter-agent edges E_{inter} represent potential communication channels. In our experiments, these edges are optimized by REINFORCE and EA, separately.

2.1.3 Node

Each node implements a fundamental operation. Our framework has 2 types of nodes; LLMNodes and AggregatorNodes.

LLMNode Makes LLM inference call to Ollama, guided by a system prompt. Each agent in the swarm is an LLMNode configured with an operation description (truthful or adversarial). The node executes asynchronously and returns the LLM’s raw text output.

MajorityVoteNode A deterministic aggregator that performs no LLM inference. Instead, it collects outputs from all predecessor nodes, applies regex-based answer extraction, and returns the most common answer. Ties are broken by random choice.

In the GPTSwarm paper and our framework, there is no node optimization process for the MMLU dataset.

2.1.4 Edge

For optimization, the active inter-agent edges are represented as a binary adjacency matrix $A \in \{0, 1\}^E$, where E is the total number of possible inter-agent edges. $A_{ij} = 1$ indicates an active edge from node i to node j ; $A_{ij} = 0$ indicates the edge is inactive.

2.2 Edge Optimization

2.2.1 REINFORCE

The original GPTSwarm framework optimizes communication topology with Reinforce [1], a policy-gradient method that operates on a continuous relaxation of the discrete edge structure.

Edge Representation In the implementation, each inter-agent edge is represented as a scalar weight $w \in \mathbb{R}$. The probability that an edge is active during graph execution is computed via sigmoid activation with temperature scaling:

$$p = \sigma\left(\frac{w}{\tau}\right) = \frac{1}{1 + \exp(-w/\tau)}$$

where τ is the temperature parameter. In experiments, $\tau = 1$, which produces the standard logistic sigmoid: edge activation probability changes smoothly as a function of weight, allowing the optimizer to explore intermediate probability values before committing to fully active or inactive edges. Lower values of τ would sharpen this transition, causing weights to more aggressively saturate toward 0 or 1.

Edge Sampling At each iteration, edges are sampled from their activation distribution: $a_e \sim \text{Bernoulli}(p_e)$ for each edge e . The resulting binary adjacency matrix defines which edges are active during that graph execution. With $M = 4$ samples per iteration, the gradient is estimated as the average over sampled graphs:

$$\hat{\nabla}_{\theta} J = \frac{1}{M} \sum_{i=1}^M (r_i - b) \nabla_{\theta} \log p_{\theta}(A_i)$$

This discrete sampling is necessary because edge activation is binary and non-differentiable.

Weight Updates via REINFORCE At each optimization step, edges are sampled stochastically according to their activation probabilities. After graph execution and reward computation, the REINFORCE gradient is applied:

$$g = \begin{cases} \text{advantage} \cdot \frac{1-p}{\tau} & \text{if edge sampled as active} \\ -\text{advantage} \cdot \frac{p}{\tau} & \text{if edge sampled as inactive} \end{cases}$$

This gradient encodes the desired direction: if an active edge received positive advantage, $g > 0$ increases the weight, making the edge more likely to activate in future samples. Conversely, if an inactive edge produced a positive advantage, $g < 0$ decreases the weight, making the edge less likely to activate by reinforcing its absence during a successful outcome. The term $\frac{1}{\tau}$ scales the gradient inversely with temperature, making steep sigmoids more sensitive to weight changes.

Gradients are accumulated and smoothed via the Adam optimizer:

$$w \leftarrow w + \alpha \cdot \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

where $\hat{m}$ and $\hat{v}$ are bias-corrected first and second moment estimates. Learning rate is $\alpha = 0.1$ 3.

Variance Reduction To reduce gradient variance, a running baseline b is maintained as an exponential moving average of observed rewards:

$$b \leftarrow \beta \cdot b + (1 - \beta) \cdot r$$

with decay $\beta = 0.9$. The advantage used in gradient computation is then $\text{advantage} = r - b$ instead of raw reward. This centers the reward signal around zero, reducing gradient magnitude and variance without introducing bias. Intuitively, advantage measures how much better the current step is relative to recent performance. Edges are encouraged if they produce rewards above average, discouraged if below. At test time, sampling is disabled (`sample=False`): each edge is activated deterministically by thresholding its probability at 0.5, yielding the most likely topology under the learned distribution.

2.2.2 Evolutionary Algorithm

In this paper, we replace REINFORCE optimizer with an evolutionary algorithm (EA). This algorithm operates directly on binary adjacency matrices, therefore eliminating the need for gradient computation.

General Idea Inter-agent communication is modeled as a directed acyclic graph (DAG). The composite swarm graph $G_{\mathcal{E}} = (N', E_{\mathcal{E}}, F', o')$ is formed by adding inter-agent edges $\mathcal{E}$ between individual agent graphs.

We measure the fitness of a topology as the fraction of questions answered correctly by the swarm. This serves as a black-box signal: the EA sees only input questions and output answers, with no access to gradients or internal model states.

We search this space with a simple Evolutionary Algorithm (EA). Each individual is an adjacency matrix. Each generation:

1. evaluates all individuals in parallel via `asyncio.gather`
2. selects parents by tournament
3. produces offspring through single-point crossover and bit-flip mutation
4. re-enforces the DAG constraint
5. carries the top elitism individuals unchanged into the next generation.
The best matrix found across all generations is returned.

The full procedure is summarised in Algorithm 2 and illustrated in Figure 1. The algorithm runs for G generations, after which the individual with the highest fitness observed across all generations is applied to the graph and evaluated on the held-out test split. The fitness function $f(\mathbf{A})$ directly approximates the task utility $u_{\tau}(G_{\mathcal{E}})$: both measure the fraction of MMLU questions answered correctly under a given topology. During training, f is estimated on batches of B questions, introducing noise into the fitness signal; the final accuracy is computed on the full held-out set, giving a cleaner estimate of the true utility. Compared to Reinforce, the EA avoids gradient estimation over discrete graphs and goes well with a parallel evaluation setup.

Genome representation Each individual in the population is represented as a flattened binary vector $\mathbf{g} \in \{0, 1\}^{n^2}$. The DAG constraint is enforced by zeroing all entries at positions (i, j) where $j \leq i$, retaining only upper-triangular entries as free bits. A population of P individuals is maintained at each generation, with the mention that the first generation is represented by randomly generated DAGs.

Fitness evaluation Each individual is evaluated by executing the full swarm on a batch of B questions drawn uniformly at random from the training split. Fitness is the fraction of correct answers, with values between $[0, 1]$.

Selection We apply tournament selection with tournament size k : k individuals are drawn uniformly at random from the current population, and the one with highest fitness is selected as a parent. This process is repeated independently for each parental slot required to fill the next generation. In addition, we apply **elitism**: the individual with the highest fitness in a generation will be copied without modification into the next one, regardless of crossover and mutation outcomes. This guarantees that the best-observed solution is never destroyed.

Crossover Given two parent genomes $\mathbf{g}^{(1)}$ and $\mathbf{g}^{(2)}$, we apply single-point crossover. A crossover point c is sampled uniformly from $\{1, \dots, |\mathbf{g}| - 1\}$, and one offspring is formed by taking the first part of $\mathbf{g}^{(1)}$ until the crossover point c , and appended to the second part of $\mathbf{g}^{(2)}$ after the crossover point c . The DAG constraint is then re-enforced after crossover by zeroing lower-triangular entries of the reshaped matrix.

Mutation Lastly, after crossover, any bit of the genome could be flipped from 0 to 1 or vice versa with a probability of p . Back-edges are removed to keep graphs valid DAGs, leaving us with an upper triangular matrix.

2.3 Error Handling and Robustness

2.3.1 Question and Answer Format

First, we extracted the questions and answers from the huggingface MMLU dataset to feed Ollama. System prompt, as can be seen from Table 2, deliberately tells the LLM to output only the answers in letters, but after debugging we detected that especially adversarial nodes might output the full answer choice or explain the reasoning in a CoT manner rarely. To avoid this, we extracted only the letter choice of the answer using a function `extract()` which finds the letter and returns it as the final answer.

2.3.2 Tie-Break Logic

Another design choice was deciding on the majority vote function. The paper didn't explicitly mention what the tie break logic is for same number of active truthful and adversarial nodes. First approach was prompting the aggregator node with "...when there is a tie, use your best logic to decide.". This resulted in very arbitrary decisions made by the LLM, also because input, which is the question is never fed into the aggregator node, aggregator node's role is to receive outputs of predecessors and count the winner. In the open source code of the `_execute()` function, a tie-break situation is solved with random choice. The aggregator node used in our implementation is then changed to randomly determining one of the answers if majority voting cannot decide on an answer.

2.3.3 Excluding Inactive Agents in the Graph Traversal

The GPTSwarm `.topological_sort()` method computes layers of the nodes using in-degree over active edges only. However, agent nodes had in-degree set to zero regardless of whether their outgoing edges are active. Consequently, agents whose edges were deactivated by the EA optimizer still executed LLM API calls, with their outputs discarded at the aggregator. This caused the nodes that are already pruned by the optimization algorithms to still make LLM calls, increasing the cost by almost 40%.

Algorithm 1 performs a depth-first search backwards through the composite graph, starting from the output node and following only active edges. The resulting set contains all agents whose outputs contribute to the final decision, or whose outputs reach the output node through at least one active edge path. With this failure detection algorithm, bigger graphs with more agents are validated before they collapse in fitness after latter generations and are misinterpreted as poor optimization.

Only nodes within this contributing set are included in the execution layers. While this fix does not seem to affect task correctness in isolation because the aggregator already filters inactive predecessors via `.active_predecessors()`, it had an indirect effect on accuracy. In the 7T7A EA run, the Ollama server degraded under sustained inference load after approximately 15 generations, causing all agents to return empty outputs and reducing accuracy to zero for the remainder of the run. By eliminating redundant LLM calls for inactive agents, the fix reduces server load and makes such degradation less likely, thereby allowing the EA to receive meaningful fitness signals throughout the full optimization. Token consumption decreased proportionally to the number of deactivated edges, reducing load on the local Ollama server and preventing erroneous time-out.

2.3.4 Graph Execution Layer

Error handling is implemented at multiple levels within the graph execution pipeline:

Critical Errors The exception that terminates execution is an unreachable output node, which raises a `ValueError`. This is a correctness check ensuring that the final decision node is reachable from at least one agent in the composite graph.

LLM Timeout Handling LLM inference timeouts are caught and converted to `RuntimeError` exceptions at the graph level. However, these exceptions are not propagated to the caller. Instead, they are caught and handled silently within `node.py`, allowing execution to continue without requiring all nodes to be present.

Node-Level Failures Individual node failures (e.g., empty LLM responses, parsing errors) trigger warnings via Python’s `warnings` module but do not raise

an exception. Failed nodes return empty outputs (`outputs=[]`), and execution proceeds to successor nodes with the remaining active nodes. When a subset of agents fails, a warning is printed to stdout and execution continues, as the majority vote aggregator can still produce a valid result from the remaining agents. When all agents fail, a `RunTimeError` is thrown and graph execution terminates.

Token Tracking Token usage statistics use a two-tier handler: anticipated API format inconsistencies like `KeyError`, `TypeError` are passed as they do not indicate a logic error, while any other unexpected exception is reported to stdout. This enables detection of genuine token tracking failures without false alarms from transient API format changes.

2.4 Experimental Setup

2.4.1 Dataset

Following the paper, the dataset used is the Massive Multitask Language Understanding dataset (MMLU) [6], which is a standard benchmark used to evaluate language understanding models. This dataset contains over 100k 4-choice general knowledge testing questions across 57 academic subjects. We sampled $N_{\text{TRAIN}} = 100$ questions to use in optimizer training, and $N_{\text{TEST}} = 100$ questions for evaluation. Questions were chosen in random with a fixed seed of 42 to ensure random subjects were used across the framework. No pre-processing beyond tokenisation is applied.

2.4.2 Model

All agents use LLaMA 3.1 8B served locally via Ollama. Inference is run at temperature 0.2 [3], introducing mild stochasticity to promote response diversity across agents within the same swarm.

2.4.3 Swarm Configurations

All experiments use four swarm sizes to test scalability: 1T1A, 3T3A, 5T5A, 7T7A, where T denotes truthful agents and A denotes adversarial agents.

Baselines We evaluate five distinct swarm topologies to establish performance anchors and isolate the effect of edge optimization:

- **IO (Single Agent):** One truthful agent directly answers the question. This is the absolute baseline which we try to achieve or, possibly, elevate.
- **Fully Connected (FC):** All agents are connected to the aggregator. No pruning of the edges. Establishes the ceiling performance without optimization.

- **Randomly Connected (RC):** Edges sampled uniformly with probability of 0.5 from E_{inter} . Controls for whether optimization better than random search.
- **REINFORCE-Optimized:** Trained for 200 steps with batch size 4, learning rate 0.1, and baseline decay 0.9.
- **EA-Optimized:** Population size 10, 15 generations, mutation rate 0.2, tournament size 2, batch size 20.

Hardware Experiments are run on the Hábrók HPC cluster using a single NVIDIA A100 GPU node (gpulong partition, 72-hour walltime at max) through 5 jobs: 1T1A, 3T3A, 5T5A, 7T7A which ran in parallel, and a aggregator job to plot graphs after all four jobs terminated.

3 Results

3.1 Ablation Study

The single-agent IO baseline achieves 46–51% accuracy depending on configuration. The fully connected swarm matches or slightly exceeds IO accuracy in every configuration (49–55%), while the randomly connected swarm performs consistently below IO (30–46%), confirming that arbitrary connectivity is harmful in the adversarial setting. Both optimizers recover accuracy relative to the random baseline: in 3T3A, REINFORCE reaches 50% and EA reaches 51%, both substantially outperforming the random baseline (30%). This pattern holds across all swarm sizes; REINFORCE and EA cluster within 1-2% of each other and of the fully connected ceiling, while random connectivity degrades performance by 15-19% relative to IO (Figure 2).

Token costs grow with swarm size for all methods while accuracy gains remain marginal (Figure 3). In 3T3A, EA consumes 4.6M tokens to reach 51% accuracy compared to 460K for REINFORCE at 50%, a $10\times$ cost increase for a 1 % gain. This relationship is consistent across configurations.

3.2 REINFORCE vs. EA Convergence

The rolling average reward of REINFORCE over 200 training steps stabilises in the range 0.30–0.50 within the first 25 steps and does not exhibit a clear monotone increase later, consistent with the high-variance behaviour. The 3T3A configuration produces the highest and most variable rewards; 5T5A and 7T7A converge to a plateau near 0.40 (Figure 4).

The best-individual fitness per generation of the EA oscillates in the range 0.25–0.80 across 15 generations without a consistent upward trend. For 1T1A the final generation achieves 0.80, the highest single-generation value observed; for 3T3A and 5T5A, best-fitness values fluctuate between approximately 0.30 and 0.70. The 7T7A run shows progressive degradation after generation 12 (Figure 5).

3.3 Learned Edge Selection: REINFORCE vs. EA

REINFORCE assigns probabilities near 1.0 to a subset of edges and near 0.0 to others at convergence. In 1T1A, the truthful edge reaches probability 1.00 while the adversarial edge is suppressed to 0.01. At larger swarm sizes (5T5A, 7T7A), REINFORCE activates 6/10 and 5/14 edges respectively but includes adversarial edges with probabilities up to 0.98 (Figure 6).

Comparing REINFORCE and EA edge selections directly (Figures 7 and 8), both methods agree in 1T1A: T0 is retained, A0 is pruned. In 3T3A they diverge: REINFORCE activates T0, T1, and A1, while EA selects T1, T2, and A2. Neither method achieves complete adversarial suppression at 5T5A or 7T7A.

3.4 Token Efficiency

EA token cost grows substantially faster than REINFORCE across swarm sizes, reaching 10.8M tokens at 7T7A versus 1.1M for REINFORCE, approximately a $10\times$ difference at the largest configuration and $3.7\times$ at 3T3A (Figure 9). This is attributable to the EA evaluating a population of 10 individuals over 15 generations with batch size 20, yielding up to $10 \times 15 \times 20 = 3000$ LLM queries during training, compared to $200 \times 4 = 800$ for REINFORCE.

The IO baseline achieves the highest token efficiency (0.0228 accuracy/1k tokens) across all configurations. Among multi-agent methods, the fully connected swarm is the most efficient (avg. 0.00379), outperforming random connectivity (0.00158), REINFORCE (0.00126), and EA (0.00013). EA is the least efficient method across all swarm sizes, spending approximately $180\times$ more tokens per unit of accuracy than a simple IO agent at 7T7A (Figures 10 and 11).

Appendix

Individual Contributions

Category	Task	Akkanat	Turak	Gavat
Implementation	README and requirements	40%	30%	30%
	Coding baseline	25%	60%	15%
	Coding modification (EA)	25%	10%	65%
	Error handling and robustness fixes	40%	45%	15%
	Experiment pipeline (SLURM/Habrok)	20%	65%	15%
Analysis	Results aggregation	50%	25%	25%
	Visualizations and figures	60%	20%	20%
	Statistical interpretation	45%	25%	30%
	Edge selection analysis	30%	30%	40%
Report	Abstract	33%	33%	34%
	Introduction	30%	40%	30%
	Methodology	20%	20%	60%
	Results	40%	20%	40%
	Discussion and limitations	35%	35%	30%
	Proofreading	33%	33%	34%
Presentation	Slide design and figures	35%	30%	35%
	Presenting / Q&A preparation	34%	33%	33%

Table 1: Estimated individual contributions across implementation, analysis, report, and presentation tasks. Each row reflects the primary contributor(s) for that task; overall workload is balanced across the three team members.

Component	Prompt
System	You are a knowledgeable expert in question answering. I will ask you a question. I will also give you 4 answers enumerated as A, B, C and D. Only one answer out of the offered 4 is correct. You must choose the correct answer to the question. Your response must be one of the 4 letters: A, B, C or D, corresponding to the correct answer. Only one letter (A, B, C or D) is allowed in your answer.
Truthful Agent	{input}
Adversarial Agent	Answer a lie to the following question: {input}.

Table 2: System and user prompts for truthful and adversarial agents in MMLU experiments. The system prompt is identical for both agent types; only the user template differs.

Category	Parameter	Value
Model	Base LLM	Llama 3.1:8b
Dataset	N_TRAIN	100
	N_TEST	100
	Seed	42
REINFORCE	Steps	200
	Batch size	4
	Learning rate	0.1
	Baseline decay	0.9
EA	Generations	15
	Population size	10
	Mutation rate	0.2
	Tournament k	2
	Batch size	20
Swarm	Configurations	1T1A, 3T3A, 5T5A, 7T7A

Table 3: Experimental hyperparameters. All REINFORCE and Swarm parameters were set to match the exact values in the paper.

Config	Method	Accuracy	Tokens
1T1A	IO (single agent)	0.51	22,353
	Fully Connected	0.52	64,328
	Random Connected	0.37	105,939
	REINFORCE	0.49	173,830
	EA	0.48	1,528,122
3T3A	IO (single agent)	0.49	21,531
	Fully Connected	0.49	145,553
	Random Connected	0.30	270,184
	REINFORCE	0.50	460,291
	EA	0.51	4,596,972
5T5A	IO (single agent)	0.51	22,353
	Fully Connected	0.50	234,079
	Random Connected	0.46	445,707
	REINFORCE	0.49	762,399
	EA	0.50	7,591,348
7T7A	IO (single agent)	0.49	21,531
	Fully Connected	0.50	314,584
	Random Connected	0.41	612,795
	REINFORCE	0.50	1,053,773
	EA	0.50	10,772,778

Table 4: Test-set accuracy and total token consumption across all configurations and methods. N_TEST = 100, seed = 42.

Swarm config	#Nodes	#Potential edges
1T1A	2	4
3T3A	6	36
5T5A	10	100
7T7A	14	196

Table 5: Swarm configuration statistics. #Nodes excludes the final decision aggregator. #Potential edges is the total number of edges subject to optimization.

Algorithm 1 Compute Active Contributing Nodes

Require: Output node o , active predecessors $\text{pred}_{\text{active}}(n)$ for each node n

Ensure: Set of nodes that contribute to final output via active edges

```
contributing  $\leftarrow \emptyset$ 
stack  $\leftarrow [o]$ 
while stack  $\neq \emptyset$  do
   $n \leftarrow \text{stack.pop}()$ 
  if  $n \notin \text{contributing}$  then
    contributing  $\leftarrow \text{contributing} \cup \{n\}$ 
    for each  $p \in \text{pred}_{\text{active}}(n)$  do
      stack.push( $p$ )
    end for
  end if
end while
return contributing
```

Algorithm 2 EA pseudocode

```
1: Initialise 10 random DAG adjacency matrices
2: for each of 20 generations do
3:   Evaluate all individuals in parallel on a batch of 20 questions
4:   Update the global best topology if improved
5:   Carry the best individual into the next generation unchanged
6:   while next generation is not full do
7:     Select parents by tournament
8:     Produce one offspring via crossover and mutation ( $p = 0.1$ )
9:     Re-enforce the DAG constraint and add to next generation
10:  end while
11: end for
12: return best topology observed across all generations
```

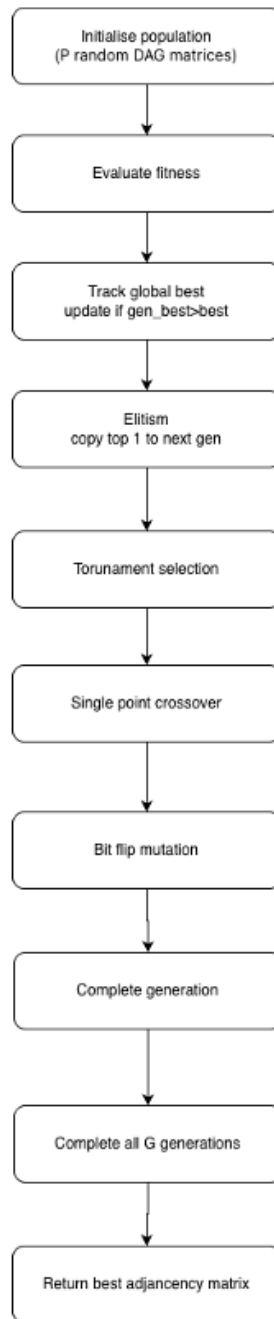


Figure 1: Evolutionary Algorithm Pipeline

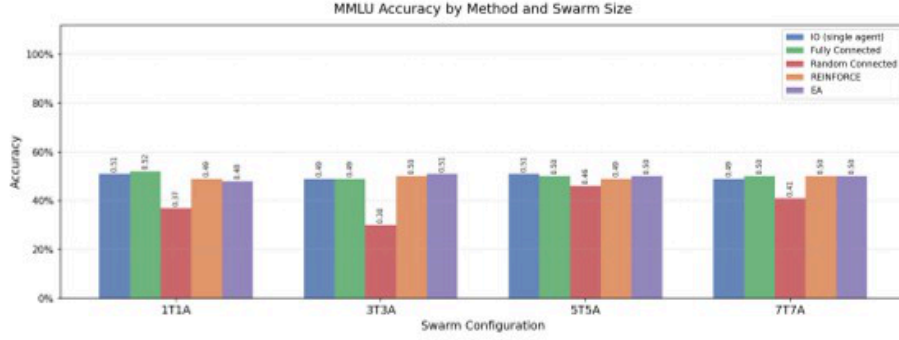


Figure 2: MMLU accuracy by method and swarm configuration (N-TEST = 100). Both REINFORCE and EA recover from the random connectivity baseline and approach the fully connected ceiling across all swarm sizes.

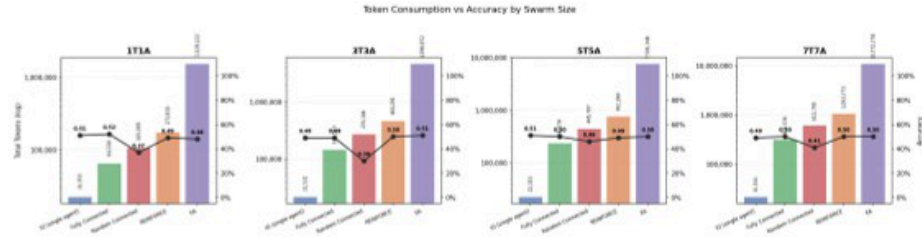


Figure 3: Token consumption (log scale, bars) and test accuracy (line, right axis) per method across swarm configurations. Token costs grow steeply for both optimizers while accuracy differences remain within 1–2 percentage points.

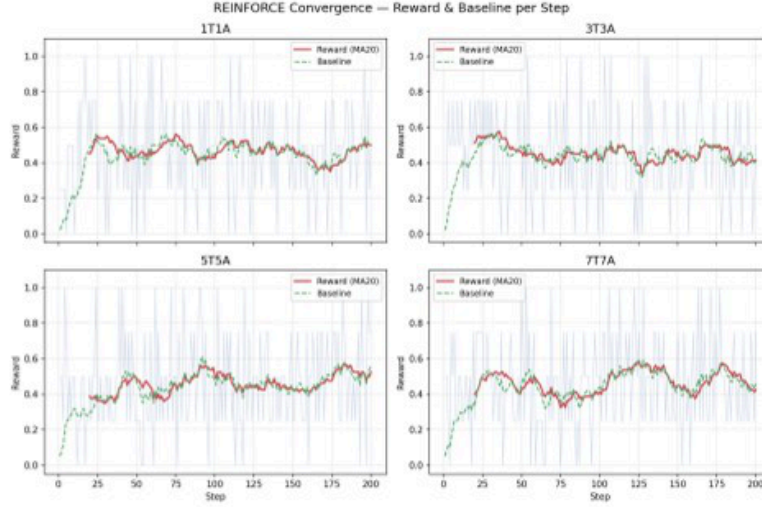


Figure 4: REINFORCE reward convergence: rolling average batch reward (window = 10 steps) across four swarm configurations. All conditions converge to a plateau near 0.40–0.50 with persistent variance; no monotone improvement is observed beyond step 25.

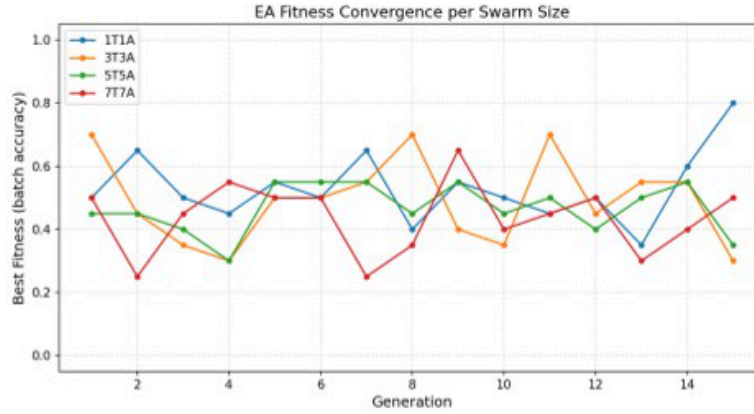


Figure 5: EA fitness convergence: best individual fitness per generation across four swarm configurations. Fitness is measured as batch accuracy over 20 questions. The 7T7A run degrades after generation 12 due to LLM server saturation.

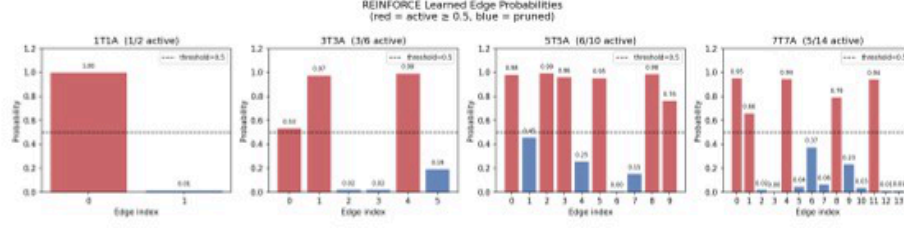


Figure 6: REINFORCE learned edge probabilities at convergence across swarm configurations. Red bars: active edges (≥ 0.5); blue bars: pruned edges. Adversarial edges remain active in 5T5A and 7T7A configurations.

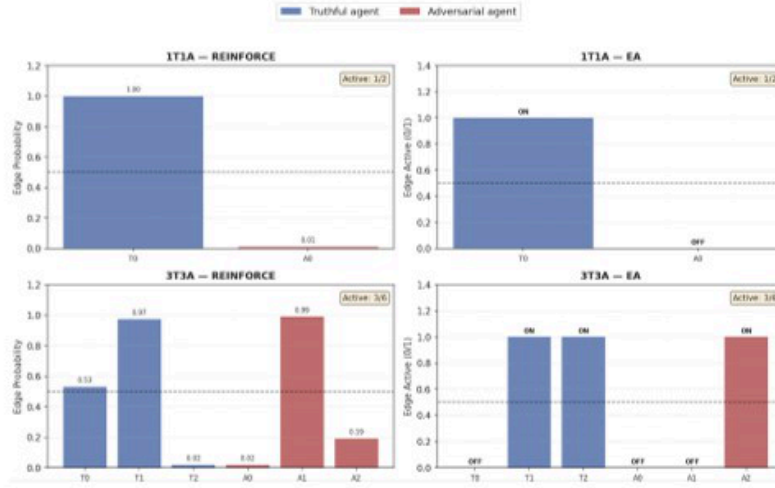


Figure 7: Learned edge selection: REINFORCE (left, continuous probability) vs. EA (right, binary active/inactive) for 1T1A and 3T3A. Blue: truthful agent edges; red: adversarial agent edges. Dashed line marks the 0.5 threshold for REINFORCE.

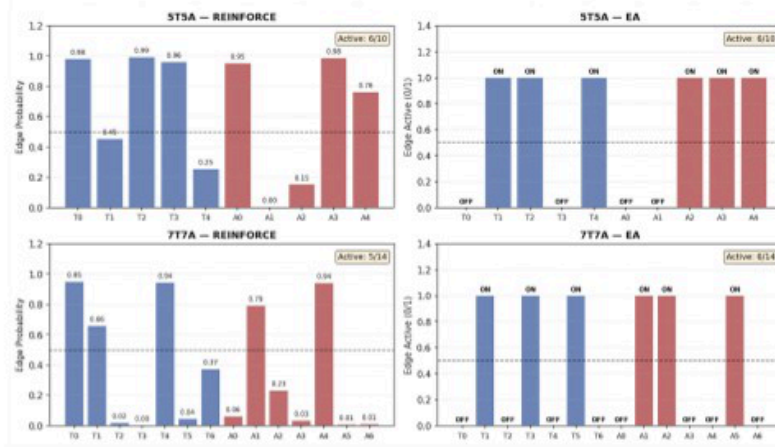


Figure 8: Learned edge selection: REINFORCE vs. EA for 5T5A and 7T7A. Both methods include adversarial edges at these scales; neither achieves full adversarial suppression.

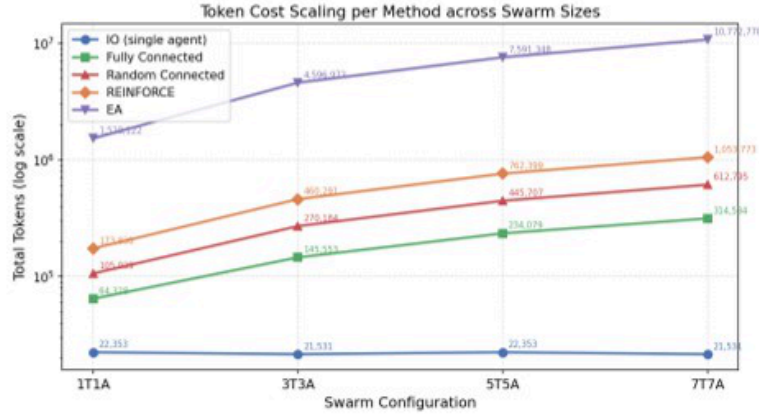


Figure 9: Token cost scaling: total token consumption per method across swarm sizes (log scale). EA cost grows substantially faster than REINFORCE; both optimizers are orders of magnitude more expensive than non-optimized baselines at equivalent accuracy.

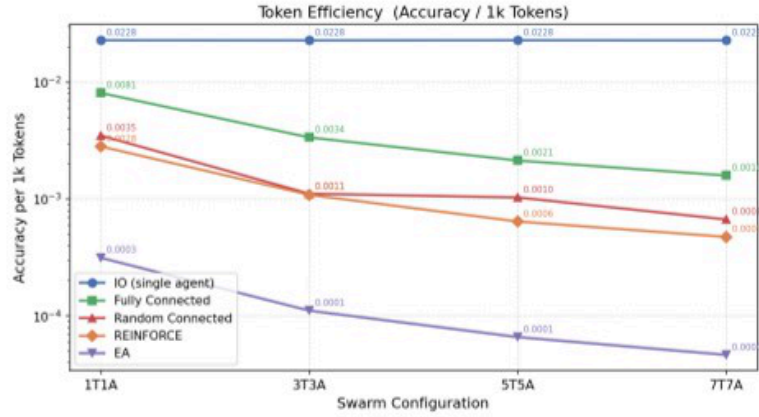


Figure 10: Token efficiency (accuracy / 1k tokens, log scale) as a function of swarm configuration. All multi-agent methods degrade in efficiency with swarm size; EA degrades most rapidly.

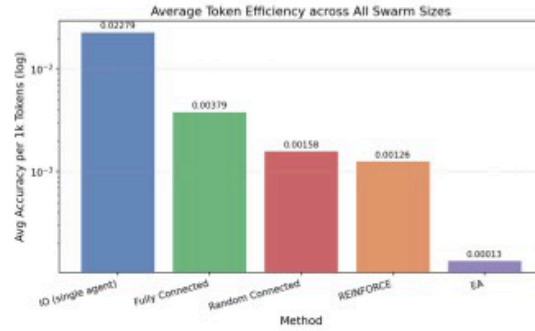


Figure 11: Average token efficiency across all swarm sizes per method (log scale). EA is the least efficient optimizer by over an order of magnitude relative to REINFORCE.

References

- [1] M. Zhuge, W. Wang, L. Kirsch, F. Faccio, D. Khizbullin, and J. Schmidhuber, “GPTSwarm: Language Agents as Optimizable Graphs,” in *Proc. 41st Int. Conf. Machine Learning (ICML)*, Vienna, Austria, PMLR 235, 2024. Available: <https://arxiv.org/abs/2402.16823>
- [2] Q. Wu, G. Bansal, J. Zhang, Y. Wu, S. Zhang, E. Zhu, B. Li, L. Jiang, X. Zhang, and C. Wang, “AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation Framework,” *arXiv preprint arXiv:2308.08155*, 2023.
- [3] S. Hong, X. Zheng, J. Chen, Y. Cheng, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, et al., “MetaGPT: Meta Programming for Multi-Agent Collaborative Framework,” *arXiv preprint arXiv:2308.00352*, 2023.
- [4] Z. Liu, Y. Zhang, P. Li, Y. Liu, and D. Yang, “Dynamic LLM-Agent Network: An LLM-Agent Collaboration Framework with Agent Team Optimization,” *arXiv preprint arXiv:2310.02170*, 2023.
- [5] R. J. Williams, “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning,” *Machine Learning*, vol. 8, no. 3–4, pp. 229–256, 1992.
- [6] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt, “Measuring Massive Multitask Language Understanding,” *CoRR*, vol. abs/2009.03300, 2020. Available: <https://arxiv.org/abs/2009.03300>
- [7] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*, 2nd ed. Berlin: Springer, 2015.