MAPRO: Recasting Multi-Agent Prompt Optimization as Maximum a Posteriori Inference

Zheyuan Zhang[1], Lin Ge[3],CR Hongjiang Li[3], Weicheng Zhu[3], Chuxu Zhang[2], Yanfang Ye[1]†
[1]University of Notre Dame, [2]University of Connecticut, [3]Amazon
†Corresponding Author
{zzhang42, yye7}@nd.edu,

Abstract

Large language models (LLMs) have demonstrated remarkable capabilities across diverse tasks, and LLM-based agents further extend these abilities to various practical workflows. While recent progress shows that multi-agent systems (MAS) can outperform single agents by coordinating specialized roles, designing effective MAS remains difficult due to prompt sensitivity and the compounded instability MAS creates. To cope with the challenge, recent efforts in automated prompt design have reduced manual effort. However, multi-agent prompt optimization remains largely unexplored. Challenges like exponentially expanding search space and ambiguous credit assignment together make systematic design intractable without principled methods. Therefore, we introduce **M**ulti-**A**gent **PR**ompt **O**ptimization (**MAPRO**), a four-stage framework that first formulates MAS prompt optimization as a *Maximum a Posteriori* (MAP) inference problem and solves it using a language-guided variant of max-product belief propagation algorithm. To address credit assignment and updates the system iteratively, MAPRO employs a topology-aware refinement mechanism that integrates execution feedback and downstream blames to selectively update agent prompts. Through this process, MAPRO progressively converges to a coordinated set of agent-specific prompt policies. Across benchmarks in various tasks, MAPRO achieves state-of-the-art performance, consistently surpassing manually engineered baselines and recent automated alternatives. Beyond performance, our MAP-based formulation also delivers general guidelines for building more reliable and principled multi-agent systems in the future [1].

1 Introduction

Large language models (LLMs) have emerged as powerful general-purpose learners, excelling at tasks that require reasoning, comprehension, and text generation. Their rapid progress has reshaped both research and practice across domains ranging from scientific discovery to software development (Kojima et al., 2022; Ouyang et al., 2022). Building upon this foundation, LLM-based agents have gained prominence for their ability to autonomously plan, interact, and solve complex problems with minimal human supervision. Such agents extend the reach of LLMs into practical workflows, enabling applications such as program synthesis and debugging, retrieval-augmented generation, data-centric analysis and interactive decision-making (Jimenez et al., 2024; Singh et al., 2025; Guo et al., 2024b; Li et al., 2024). While single agents are useful, orchestrating multiple LLM agents in a coordinated system has shown even greater promise (Hong et al., 2024a; Wang et al.,

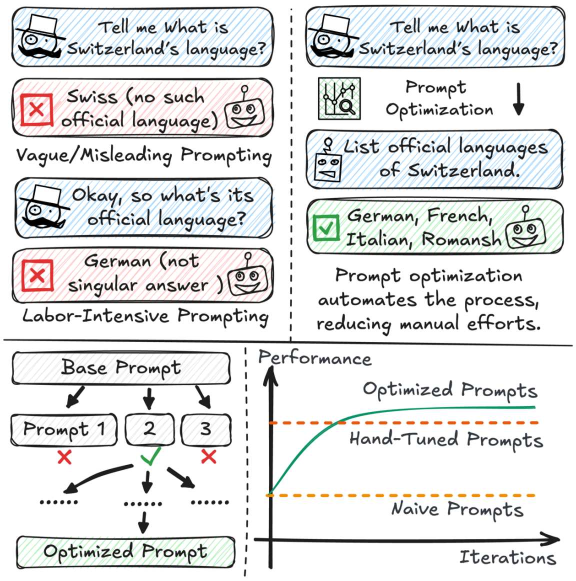
Tell me What is Switzerland's language?
Swiss (no such official language)
Vague/Misleading Prompting
Okay, so what's its official language?
German (not singular answer)
Labor-Intensive Prompting
Tell me What is Switzerland's language?
Prompt Optimization
List official languages of Switzerland.
German, French, Italian, Romansh
Prompt optimization automates the process, reducing manual efforts.
Base Prompt
Prompt 1
2
3
......
......
......
Optimized Prompt
Performance
Optimized Prompts
Hand-Tuned Prompts
Naive Prompts
Iterations

Figure 1: Prompt quality governs agent reliability: (top-left) vague, manually revised prompts are error-prone and costly; (top-right) automated prompt optimization search and produce correct answers; (bottom-left) the optimizer explores and selects among candidate rewrites; (bottom-right) performance improves over iterations, surpassing hand-tuned prompting.

[1]Work done during internship at Amazon.

2024b). Multi-agent systems (MAS) leverage diverse perspectives and roles, such as critics, verifiers, or debaters, to collectively outperform single-agent counterparts in reasoning, exploration, and robustness (Yuan et al., 2024; Shinn et al., 2023; Qian et al., 2025; Wang et al., 2025). Yet, constructing effective MAS is far from straightforward. A recurring difficulty lies in prompt sensitivity, where small variations in instructions can drastically alter behavior and degrade performance (Zhou et al., 2024). In multi-agent settings, where outputs cascade across agents, such fragility may be compounded, amplifying instability across the system (Zhou et al., 2025).

To mitigate these challenges, recent work has explored various forms of automated prompt design and system adaptation. Broadly, these approaches aim to reduce reliance on manual engineering by algorithmically refining prompts, adjusting agent roles, or restructuring interaction patterns (Khattab et al., 2024; Hu et al., 2025). However, despite these advances, the problem of prompt optimization in multi-agent settings remains largely underexplored. This gap arises from two core challenges: (1) the search space grows combinatorially as each agent maintains its own set of prompt candidates, making it extremely difficult to navigate efficiently and leaving the system vulnerable to suboptimal local choices rather than coordinated global improvement; (2) credit assignment is highly uncertain, since it is rarely clear which agent's prompt should be targeted for refinement, how it should be modified, or whether adjustments at the individual level will even translate into system-wide gains.

To tackle these challenges, in this paper, we propose **M**ulti-**A**gent **PR**ompt **O**ptimization (**MAPRO**), a four-stage framework that jointly explores the multi-agent prompt space, propagates feedback signals, and iteratively refines prompt policy of each agent. By grounding optimization in a principled inference process, MAPRO provides a structured approach for navigating the otherwise intractable combinatorial landscape of MAS design. Specifically, to cope with the exponential search space, we formalize multi-agent prompt optimization as a *Maximum a Posteriori* (MAP) inference problem over Directed Acyclic Graphs (DAGs), and develop a language-guided variant of the max-product belief propagation (MPBP) algorithm. This design leverages agent-level and interaction-level reward models to efficiently approximate globally optimal prompt assignments in polynomial time complexity. Furthermore, to address the inherent ambiguity in credit assignment, MAPRO introduces a topology-aware refinement procedure that maintains distinct prompt policies for each agent rather than collapsing the system into a single global policy. By distributing credit by incorporating the blames from downstream agents, MAPRO progressively converging toward a set of coordinated yet agent-specific prompt policies that enhance overall system robustness and performance. Through iterative optimization, MAPRO produces multi-agent systems that achieve state-of-the-art performance, surpassing both manually engineered MAS baselines and automatically generated alternatives in single- and multi-agent settings. These improvements are consistently demonstrated across diverse tasks, including mathematical reasoning, question answering, and code generation. Our contributions can be summarized as follows:

- **MAP Inference Formulation.** To our best knowledge, we are the first to cast multi-agent prompt optimization as a *Maximum a Posteriori* (MAP) inference problem. This formulation provides a principled objective for navigating the combinatorial search space, enables efficient approximation of globally optimal prompt sets, and offers general guidelines for systematic prompt optimization design.

- **Topology-aware Credit Assignment.** We propose a novel refinement mechanism that integrate execution feedback and downstream blames, which alleviate the challenge of ambiguous credit assignment, enabling targeted improvements to specific agents with distinct prompt policies.

- **State-of-the-Art Performance.** On diverse benchmarks—including mathematical reasoning, question answering, and code generation—MAPRO consistently surpasses manually engineered MAS baselines and recent automated alternatives, establishing new state-of-the-art results in multi-agent prompt optimization.

2 Preliminary

2.1 Multi-agent System as Directed Graph

We study a *multi-agent system* (MAS) composed of N large-language-model agents that collaborate on a shared code-generation workflow. Let the index set of agents be $\mathcal{A} = \{1, \ldots, N\}$. For every agent $i \in \mathcal{A}$, the agent-specific *prompt candidate pool* is defined as

$$P_i = \{p_i^1, p_i^2, \ldots, p_i^K\}, \quad (1)$$

where $p_i^{(k)}$ is the k-th candidate prompt ($k = 1, \ldots, K$) and K is the uniform pool size. For clarity, we denote by p_i^* the optimal prompt candidate, while $\tilde{p}_i$ represents a selected prompt candidate drawn from the candidate pool. Because collaboration unfolds through directed hand-offs of textual outputs, we encode these dependencies as a directed graph $G = (\mathcal{V}, \mathcal{E}), \mathcal{V} = \mathcal{A}$, in which each vertex $i \in \mathcal{V}$ corresponds to agent i, and a directed edge $(i, j) \in \mathcal{E}$ signifies that the output of agent i is consumed as (part of) the input of agent j. This graph abstraction concisely captures the information-flow topology that underpins the subsequent optimization problem.

2.2 MAS Prompt Optimization

A prompt set $\tilde{P} = (\tilde{p}_1, \ldots, \tilde{p}_N)$ is considered successful if the entire workflow executed successfully and correctly. Unlike single-agent settings, failures in MAS stem from two sources: **1) Agent Incompetence**—producing incorrect code even from well-formed input, thereby propagating errors; and **2) Defective Interaction** — an upstream agent returning semantically irrelevant text that blocks downstream progress. Both hazards need to be properly addressed to achieve good performance.

To make these notions quantitative, we define the *agent score* to record the empirical quality of the k-th prompt of agent i as $\boldsymbol{g(p_i^k)}$, and the *edge score* to measure the reliability of the corresponding hand-off between the k-th prompt of agent i and the l-th prompt of agent j as $\boldsymbol{g(p_i^k, p_j^l)}$. Both measures lie in $[0, 1]$, where the value 1 denotes flawless behavior. To maximize the system performance and to reflect that the overall workflow is only as reliable as its weakest link, we propose the *Joint Quality Score* for the multi-agent system as:

$$\mathcal{T}(\tilde{P}) = \Big(\prod_{i=1}^{N} g(\tilde{p}_i)\Big)\Big(\prod_{(i,j)\in\mathcal{E}} g(\tilde{p}_i, \tilde{p}_j)\Big). \quad (2)$$

Note that we put $\tilde{p}$ in the equation here and omit k and l for simplicity. Intuitively, the performance $\mathcal{T}(\tilde{P})$ of a MAS is good when every agent is competent and every hand-off is clean, because a single failure at any node or edge derails the execution. In practice, for agent i, there will be K agent scores, and the same logic applies for the edge scores as well, so for (i, j), there will be K^2 edge scores. Therefore, the objective of MAS prompt optimization can be defined as:

$$P^* = argmax_{P \in P_1 \times \cdots \times P_N} \mathcal{T}(P). \quad (3)$$

Equation (3) can be viewed as the *posterior likelihood* that the entire system completes the evaluation batch without error, conditioned on the hidden prompt set P. Indeed, if we regard each agent outcome and each edge hand-off as independent Bernoulli events given P, then under a uniform prior over prompt sets, maximizing $\boldsymbol{\mathcal{T}(P)}$ is thus equivalent to the classical *maximum-a-posteriori* (MAP) problem (Proved in Appendix-C.1)

Why is this problem challenging?

As can be seen, the brute-force search space $P_1 \times \cdots \times P_N$ contains K^N discrete combinations. Moreover, the factors are highly interdependent: changing a single prompt p_i can affect many downstream agents, making greedy or local strategies prone to failure. As the objective is non-convex, discontinuous, and combinatorial, effective optimization must exploit additional structure—here, the acyclic topology of the graph G (with a prescribed iteration limit)—to prune the search space and assign credit correctly among interacting prompts. In the next section, we present an algorithm that leverages these properties to approximate P^* in polynomial time.

3 Methodology

Now that we have formalized the optimization objective for multi-agent prompt optimization and highlighted the challenges, we proceed to detail our proposed framework, **M**ulti-**A**gent **PR**ompt **O**ptimization (**MAPRO**). MAPRO addresses the Multi-agent System (MAS) prompt-optimization problem by formulating it as a discrete *maximum-a-posteriori* (MAP) inference over the joint prompt space and solving it via an iterative, LLM-guided algorithm. In particular, our method comprises four stages: (1) Initialization of prompt candidates and reward models; (2) Language-based MAP selection, which employs LLM-based reward models to conduct max-product belief-propagation algorithm to efficiently find the optimal prompt combination (Figure-2 c.1); (3) Preference-based prompt policy update, which updates the prompt pools and reward

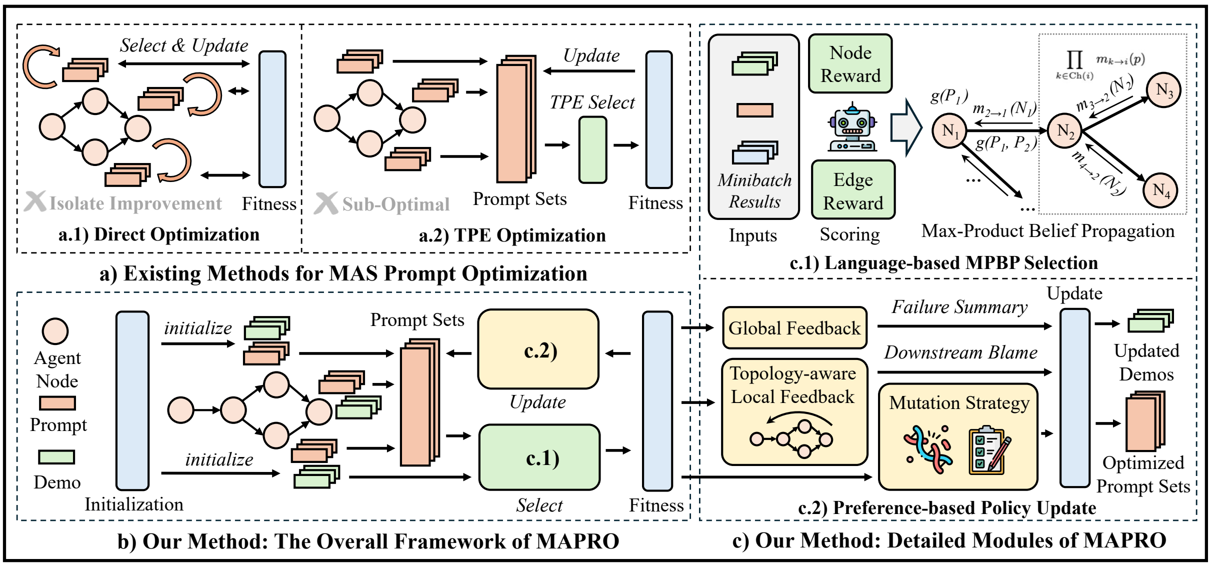

Figure 2: The Overall Framework of MAPRO. Specifically, a) shows the existing methods of prompt optimization for MAS and their drawbacks; b) shows the overall framework of MAPRO compared with existing methods; and c) demonstrate the detailed modules used in MAPRO.

models based on multi-level feedback (Figure-2 c.2); and (4) Termination, which defines stopping criteria and yields the final optimized prompt set for downstream left-out testing. We next describe each stage in detail.

3.1 Initialization

Prompt Candidate Pool Setup. MAPRO is designed as a plug-and-play setting atop any given MAS. We assume an established MAS as $G = (\mathcal{V}, \mathcal{E})$ (as defined in the preliminaries) and an initial set of base prompts $P^0 = \{p_1^0, \dots, p_N^0\}$ for the N agents. The first step is to construct a diverse prompt candidate pool P_i for each agent i by mutating its original prompt to K candidates following standard practice (Wang et al., 2024c; Xiang et al., 2025), yielding semantically similar variants $P_i = \{p_i^1,\, p_i^2,\, \dots,\, p_i^K\}$.

Preference Demonstration Pool Setup. Inspired by the reward model design of TPO (Li et al., 2025), which demonstrate human preferences can be aligned during inference without retraining and achieve comparable results, we instantiate a reward model R and seed the reward model with a set of *accepted* (positive) and *rejected* (negative) example prompt as few-shot *preference demonstrations* to guide it to generate scalar scores for each agent node and edge. Intuitively, it will judge the quality of an agent's output in isolation, and the quality of a hand-off between agents.

To initialize these examples, we first run the MAS on a mini-batch of training tasks $\mathcal{B}$ using a few random draws from the prompt pool P_i. and collect their full interaction traces if the entire task is solved correctly end-to-end on $\mathcal{B}$ Each such successful trace serves as the initial positive exemplars - the prompt p_i used is considered d^+ and the output produced can be recorded as a *desirable response* for that agent; likewise, for every edge, the message serves as an initial example of a good hand-off. To obtain complementary failure example prompts, we then generate synthetic negatives by perturbing the successful prompts as d^-. Thus we obtain, for each agent and edge, a pool of preference prompt responses $\mathcal{D} = \{d^+,\, d^-\}$. The detailed input of the reward model are provided in the section below.

3.2 Language-based MAP Selection

Given the MAP formulation of prompt optimization from above, during the action selection phase, our goal is to efficiently find the prompt assignment $P^* = \{p_1^*,\, p_2^*,\, \dots,\, p_N^*\}$ that maximizes the joint quality score. As previously discussed, directly searching the exponentially large space $P_1 \times \cdots \times P_N$ is intractable. Therefore, we exploit the factorized structure of the multi-agent system (MAS) and introduce *LLM-guided Max-Product Belief Propagation (LMPBP)*, which consists of two steps, specifically, reward model scoring and optimal prompt searching.

Reward Model Scoring. In the first stage, we prompt the reward model R to assign numerical scores between 0 to 1. For each agent i, the reward model R_i will rank each prompt $p_i^k \in P_i$ and evaluates how well the prompt would enable agent i to fulfill its role. This evaluation is conditioned on the preference demonstrations $\mathcal{D}_i$ and the corresponding input x_i and desirable response y_i:

$$g(p_i^k) = R_i(x_i; y_i; \mathcal{D}_i; P_i), \tag{4}$$

Similarly, for each directed edge $(i \rightarrow j)$, the reward model R_{ij} produces a score $g(p_i^k, p_j^l)$ reflecting how well agent i's output under prompt p_i would set up agent j for success. Concretely, we have:

$$g(p_i^k, p_j^l) = R_{ij}(y_i, \mathcal{D}_j; P_j), \tag{5}$$

This way, we have obtained the reward scores for factors required in the searching step.

Optimal Prompt Searching After the reward scores are secured, the second stage applies LMPBP to find the global optimum $\mathcal{T}(P)$ exactly in the DAG by passing local messages that aggregate optimal sub-solutions. For MAS with multiple parent dependencies, we convert the structure to equivalent tree-structured via a junction-tree transformation (Implementation details and equivalence proof in Appendix-C.2). The message-passing process works as follows: First, it goes through a leaf-to-root pass. (Note here the notations have different meanings) Assume agent i receives messages from all of its children (downstream agents for which i is an input), and then sends an aggregated message up to its own parent j. Specifically, for each possible prompt choice of its parent, agent i computes

$$m_{i\to j}(p_j) = \max\Big[g(p_i)g(p_i, p_j) \prod_{k\in Child(i)} m_{k\to i}(p_i)\Big]. \tag{6}$$

Here *Child*(i) denotes the set of agents that depend on i's output. Intuitively, $m_{i\to j}(p_j)$ represents the best achievable joint score of the entire subtree rooted at i, given that i's parent j is fixed to prompt p_j. In other words, i considers all its own prompt options and those of its descendants, and encapsulates the optimal outcome (in terms of product of local scores) in a message indexed by p_j. Once the upward messages reach the designated root agent r (the entry point of MAS), we calculate the root belief and that agent can evaluate the total score for each of its prompt candidates using equation-6.

Algorithm 1 MAPRO Overall Process

```
Initialization: Set up prompt pools $P$, and demonstration preferences $\mathcal{D}$.
while termination condition not met, do
    // Language-based MAP Selection
    Retrieve reward scores $g(p_i)$ and $g(p_i, p_j)$.
    Upward pass to retrieve localized optimal score $m_{i\to j}(p_j)$ at each node.
    Downward pass to assign best prompt $p*$ given parents' choices.
    Run with $P^\star$ on task $\mathcal{B}$; update score $S(t)$.
    // Preference-guided Policy Update
    Update $\mathcal{D} \leftarrow Critic(\mathcal{D}; P; g(P))$.
    Get $P \leftarrow \{Mutate(\mathcal{M}(P^*), f_g, f_l), P^*\}$
    if improvements $\leq \varepsilon$ over $T$ steps then
        break
    end if
    $t \leftarrow t + 1$
end while
Inference: Freeze $P^\star$; test on held-out tasks.
```

This combines r's own goodness score with the messages from all its children (each of which already accounts for the best configuration of that child's subtree). We then select the highest-belief prompt for the root:

$$p_r^* \; = \; \arg\max \beta_r(p_r). \tag{7}$$

Finally, we perform a downward pass to fix the prompts of the remaining agents based on the root decision. We visit each child i of the root and choose the prompt that attained the maximum in $m_{i\to r}(p_r)$:

$$p_i^* = \arg\max\Big[g(p_i)\, g(p_i, p_r^*) \prod_{k\in Child(i)} m_{k\to i}(p_i)\Big]. \tag{8}$$

This gives the optimal prompt for agent i assuming the root was p_r^*. We then recursively select their best prompt given p_i^*, and so on, until all agents in the graph have an assigned prompt. This backtracking procedure propagates the optimal choices down the tree, yielding the globally optimal prompt set P^* (Proved in Appendix-C.3). This selected prompt set will next be used in the refinement stage to collect feedback and further improve the prompt pools and reward models.

3.3 Preference-based Policy Update

The MAP selection phase yields the global optimal prompt set P^*, along with an explicit assessment of each agent prompt and hand-offs via the reward scores. In the prompt policy refinement phase, we leverage this information, together with actual execution feedback or diffs on tasks, to update and improve the prompt pools and reward models. The key idea is to incorporate feedback from multiple levels: (i) global-wise execution results, (ii) downstream agent blames, and (iii) controlled prompt mutation strategy to force small edits. By integrating the multi-level feedback, we can introduce targeted prompt variations to explore new parts of the search space. After refinement, the MAS is ready to perform another round of MAP-based selection with updated components. We detail the feedback collection and update steps below.

Reward and Expected Output Update. We evaluate the performance given P^* on a set of representative tasks (e.g., the training question batch $\mathcal{B}$) and we update each agent's outputs y_i as the new *desirable response* for next cycle of updates. We then use the reward scores as standards to update the accepted and rejected prompt responses for each agent. Specifically, we use a critic LLM model to judge if for agent i, the prompt p_i^k should be updated as d^+ or d^-. Formally,

$$\mathcal{D}_i \leftarrow Critic(\mathcal{D}_i; P_i; g(P_i)). \tag{9}$$

This process make sure the preference demonstrations are continually updated so that the reward scores are more closely align with actual task success.

Prompt Pool Refinement. We improve the prompt candidate pool by generating new variations using a mutate LLM model with innovative feedback design from three aspects: global feedback f_g indicating the final execution feedback; local feedback f_l which takes the reversed topology and ask each agent to generate blames to it upstream agents based on their generated input and f_g, achieving fine-grained credit assignment; and a predefined mutation strategy set with small edits $\mathcal{M}$ which mimics the idea of *trust region* in MAP policy optimization, keep the prompt variations from drifting afar. Formally, we invoke an LLM-based prompt mutation function to produce a refined prompt pools P^{new} that modifies P^*:

$$P_i^{new} = \{Mutate(\mathcal{M}(p_i^*), f_g, f_l), p_i^*\}. \tag{10}$$

Through such prompt augmentation, the MAS explores new prompts that are informed by past failures yet remain close to proven good prompts, thereby continuously improving robustness. Finally, the updated prompt pools are then used in the next iteration of MAP-based selection.

3.4 Termination

We iterate the *select–update* loop until the improvements in the joint reward have saturated, indicating convergence to an optimal prompt policy. To formalize the stopping criterion, let $S^{(t)}$ denote the joint validation score (e.g., pass rate) obtained by the best prompt set at iteration t. We define $\Delta S_t = S^{(t)} - S^{(t-1)}$ as the improvement in reward compared to the previous iteration. We choose a fixed *patience window* size T (e.g., $T = 3$) and a small tolerance $\varepsilon \geq 0$. After each iteration $t \geq T$, collect the recent gains $\{\Delta S_{t-T+1}, \ldots, \Delta S_t\}$. and we terminate the optimization loop when

$$\max_{i=1,\ldots,T} \Delta S_{t-i+1} \leq \varepsilon, \tag{11}$$

which means no improvement exceeding ε has been observed in the last T iterations. This rule halts exactly when the system has shown no progress over the specified window, ensuring that computation stops once the prompt policy has plateaued. After termination, we obtain the final optimized prompt set P^* for test-time inference on unseen tasks. By locking in P^*, we ensure the efficiency of the system that no additional time is required during inference. The time complexity of training phase is analyzed in Appendix-C.4

4 Experiments

4.1 Experimental Setup

Benchmarks. We conduct experiments on an extensive collection of tasks: HumanEval-ET, MBPP-Plus and CodeContest for code generation task, NewsQA and WebQuesion for question answering task, and MATH and GSK8K for math reasoning task. Since we are focusing on the prompt optimization and have a training scheme, we discuss the split of sets with other details including citations of these benchmarks in Appendix-B.1.

Baselines. We consider the following types of baselines: 1) Single agents without prompt optimization, including the raw model, and the most classical baselines CoT and ReAct; 2) Single agents with prompt optimization, including two most recent SOTA baselines EvoPrompt, and PromptBreeder. 3) Classical Multi-agent baselines without prompt optimization. While there are many MAS, we hope to choose the ones that are designed for general tasks, recent SOTA and covering as many types of common topologies as possible, therefore we choose Chain Design, DMAD, and the "swarm"

Backbone: Claude Haiku 3.5			Code Generation			Question Answering		Math Reasoning	
Model	**MAS**	**Optimized**	**HumanEval-ET**	**MBPP-Plus**	**CodeContest**	**NewsQA**	**WebQuestion**	**MATH**	**GSM8K**
Raw	✗	✗	69.38 ± 3.24	70.93 ± 0.34	20.36 ± 2.29	49.12 ± 0.11	33.50 ± 0.41	59.54 ± 1.10	88.57 ± 0.53
CoT	✗	✗	70.31 ± 1.91	71.98 ± 0.39	22.91 ± 1.46	54.44 ± 0.30	33.22 ± 0.32	60.25 ± 0.56	90.71 ± 0.34
ReAct	✗	✗	72.19 ± 1.71	71.02 ± 0.31	21.21 ± 0.61	58.72 ± 0.30	33.60 ± 0.34	61.29 ± 1.03	91.50 ± 0.39
EvoPrompt	✗	✓	75.63 ± 2.10	73.97 ± 0.48	22.18 ± 1.26	60.44 ± 0.74	34.99 ± 0.39	60.81 ± 1.66	92.37 ± 0.31
PromptBreeder	✗	✓	75.31 ± 0.70	74.13 ± 0.22	21.45 ± 1.47	60.76 ± 0.17	35.12 ± 0.51	60.43 ± 0.52	92.24 ± 0.18
Chain	✓	✗	71.88 ± 1.10	74.34 ± 1.14	28.85 ± 2.29	60.88 ± 0.72	34.85 ± 0.40	62.82 ± 0.89	92.06 ± 0.27
w/t Direct	✓	✓	73.96 ± 2.30	74.87 ± 0.53	29.70 ± 0.61	62.20 ± 1.31	34.25 ± 0.21	63.80 ± 0.21	92.76 ± 0.14
w/t TPE	✓	✓	75.00 ± 1.56	75.22 ± 0.40	29.90 ± 0.93	63.80 ± 0.20	34.01 ± 0.13	62.81 ± 0.37	92.72 ± 0.21
w/t MAPRO	✓	✓	**80.21 ± 0.90**	**76.54 ± 0.67**	31.52 ± 0.61	**64.00 ± 0.35**	34.65 ± 0.30	**64.30 ± 0.59**	**93.48 ± 0.42**
DMAD	✓	✗	72.19 ± 1.31	73.02 ± 0.37	36.77 ± 0.93	60.40 ± 0.37	34.43 ± 0.44	61.08 ± 0.39	90.39 ± 0.46
w/t Direct	✓	✓	73.44 ± 1.56	74.07 ± 0.27	38.79 ± 0.61	62.20 ± 0.20	34.91 ± 0.07	62.85 ± 0.11	91.19 ± 0.36
w/t TPE	✓	✓	72.92 ± 2.39	73.54 ± 0.53	37.58 ± 0.61	61.80 ± 0.35	35.22 ± 0.13	62.81 ± 0.37	91.87 ± 0.29
w/t MAPRO	✓	✓	77.08 ± 1.81	74.60 ± 0.27	38.99 ± 1.95	62.93 ± 0.12	**35.50 ± 0.33**	63.33 ± 0.46	91.96 ± 0.58
ChatEval (Swarm)	✓	✗	73.44 ± 1.10	72.60 ± 0.34	38.79 ± 1.05	60.36 ± 0.26	33.33 ± 0.34	62.62 ± 0.97	91.59 ± 0.89
w/t Direct	✓	✓	74.48 ± 2.38	73.19 ± 0.46	38.18 ± 0.61	61.80 ± 0.20	34.17 ± 0.29	63.83 ± 0.84	91.42 ± 0.47
w/t TPE	✓	✓	76.04 ± 0.90	73.28 ± 0.26	40.61 ± 0.61	62.53 ± 0.31	34.35 ± 0.18	62.68 ± 0.58	91.31 ± 0.62
w/t MAPRO	✓	✓	78.13 ± 1.56	73.98 ± 0.15	**41.41 ± 0.93**	62.67 ± 0.31	34.52 ± 0.20	63.13 ± 0.84	91.73 ± 0.41

Table 1: Performance results with baseline methods on Claude Haiku 3.5. We report the mean and standard deviation for all results. The best performance is bolded and runner-ups are underlined.

Prompt Optimization Example

Base instruction:

You are a Python programmer. Write pure, runnable Python code that solves the task.

Adding:

You are a Python programmer. Write pure, runnable Python code that solves the task. **Ensure the solution is a single function named solution with robust input validation**, direct implementation, and **no type hints**. **Handle edge cases** explicitly and provide clear, executable code.

Replacement:

You are a Python programmer. Write pure, runnable Python code that solves the task. Ensure the solution is a single function named solution with **robust input validation using isinstance() checks, type conversion fallbacks**, and comprehensive error handling. **Use try-except blocks with specific exception types, provide default values for edge cases**, . . .

Table 2: "Adding" appends guidance to the prompt while "Replacement" rewrites previous parts with better instructions. Colors highlight the modified texts.

version of ChatEval, which is Simultaneous-Talk-with-Summarizer. 4) Prompt optimization for MAS baselines. Since we only consider prompt optimization[2], we adopt the direct optimization method from GPTSwarm and Tree-structured Parzen Estimator (TPE) methods used in MASS and MIPRO to each of the MAS we described above to make a comprehensive comparison. More details including the citations of the baselines are in Appendix-B.2.

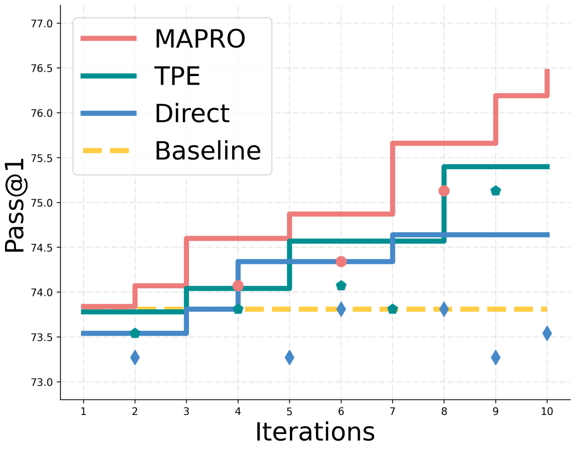

Figure 3: Optimization trajectories on the MBPP+ benchmark. We report the first ten optimization iterations using the chain MAS framework. MAPRO exhibits a more consistent and steady improvement compared to alternative methods.

4.2 Main Results

We present the main results of MAPRO against baselines in Table 1 and Table 3. Across all benchmarks, MAPRO consistently achieves superior performance, often setting the best results within the same MAS, underscoring the strength of our approach. Several additional insights emerge. In terms of MAS structure, while topology exerts a stronger influence on overall accuracy than prompts, our plug-and-play design offers unmatched flexibility and extensibility, avoiding the heavy cost of topology optimization and enabling efficient deployment. As for task characteristics, MAPRO delivers the largest gains on reasoning-intensive tasks (e.g., WebQuestions, MBPP-Plus) compared to knowledge-heavy ones (e.g., NewsQA, CodeContest), highlighting the unique advantage of prompt optimization in com-

[2] In this paper, we consider the plug-and-play settings an unique advantage and don't update the topology because this is more friendly to industry scenarios where teams already have their developed MAS implemented and in production.

Backbone: Llama 3.3-70b			Code Generation			Question Answering		Math Reasoning	
Model	**MAS**	**Optimized**	**HumanEval-ET**	**MBPP-Plus**	**CodeContest**	**NewsQA**	**WebQuestion**	**MATH**	**GSM8K**
Raw	✗	✗	67.81 ± 0.86	68.04 ± 0.68	19.76 ± 2.08	58.65 ± 0.96	33.15 ± 0.50	67.56 ± 1.44	91.71 ± 0.37
CoT	✗	✗	68.44 ± 1.31	68.04 ± 0.22	21.09 ± 1.31	60.56 ± 0.35	34.35 ± 0.47	69.01 ± 0.66	92.06 ± 0.29
ReAct	✗	✗	69.06 ± 0.70	68.15 ± 0.44	20.36 ± 0.69	62.06 ± 0.28	35.84 ± 0.54	69.26 ± 0.84	92.34 ± 0.27
EvoPrompt	✗	✓	72.19 ± 1.71	69.74 ± 0.64	20.85 ± 1.33	64.37 ± 0.34	35.47 ± 0.41	71.62 ± 0.38	93.12 ± 0.19
PromptBreeder	✗	✓	71.88 ± 1.10	69.10 ± 0.29	20.85 ± 1.40	64.53 ± 0.43	35.77 ± 0.16	71.01 ± 0.44	93.05 ± 0.23
Chain	✓	✗	70.00 ± 1.31	68.68 ± 0.14	27.52 ± 2.08	63.20 ± 0.45	35.24 ± 0.28	71.45 ± 1.04	93.71 ± 0.25
w/t Direct	✓	✓	71.35 ± 1.80	69.58 ± 0.46	28.08 ± 0.70	63.62 ± 0.27	36.16 ± 0.18	70.37 ± 0.57	93.89 ± 0.31
w/t TPE	✓	✓	71.88 ± 1.56	70.28 ± 0.40	28.69 ± 0.93	63.80 ± 0.21	36.04 ± 0.03	71.64 ± 0.62	93.42 ± 0.22
w/t MAPRO	✓	✓	**75.00 ± 1.56**	**72.31 ± 0.55**	30.10 ± 0.70	63.82 ± 0.54	36.22 ± 0.30	71.87 ± 0.35	93.56 ± 0.34
DMAD	✓	✗	70.94 ± 0.86	70.37 ± 0.46	34.06 ± 0.90	63.82 ± 0.54	35.56 ± 0.11	69.85 ± 0.42	94.12 ± 0.19
w/t Direct	✓	✓	71.88 ± 1.56	70.90 ± 0.26	35.35 ± 0.70	64.12 ± 0.35	36.02 ± 0.20	70.99 ± 0.33	94.93 ± 0.33
w/t TPE	✓	✓	71.35 ± 1.80	70.55 ± 0.40	34.55 ± 0.61	64.30 ± 0.29	36.14 ± 0.22	71.32 ± 0.26	94.67 ± 0.28
w/t MAPRO	✓	✓	73.96 ± 0.90	71.60 ± 0.31	35.96 ± 1.53	65.10 ± 0.24	36.22 ± 0.28	**72.99 ± 0.33**	**95.91 ± 0.30**
ChatEval (Swarm)	✓	✗	71.25 ± 0.86	71.43 ± 0.19	34.91 ± 1.01	63.73 ± 0.43	36.44 ± 0.32	71.32 ± 0.26	93.14 ± 0.31
w/t Direct	✓	✓	72.92 ± 0.90	70.99 ± 0.31	35.15 ± 0.61	64.02 ± 0.26	36.36 ± 0.27	71.73 ± 0.57	92.58 ± 0.37
w/t TPE	✓	✓	72.40 ± 2.39	71.78 ± 0.31	36.36 ± 0.61	64.18 ± 0.31	36.41 ± 0.27	71.48 ± 0.80	92.45 ± 0.34
w/t MAPRO	✓	✓	**75.00 ± 1.56**	72.22 ± 0.26	**37.17 ± 0.93**	**65.45 ± 0.09**	**36.55 ± 0.29**	72.26 ± 0.71	92.38 ± 0.40

Table 3: Performance results with baseline methods on Llama 3.3-70b. We report the mean and standard deviation for all results. The best performance is bolded and runner-ups are underlined.

Method	HumanEval-ET	MBPP-Plus	CodeContest	NewsQA	WebQuestion	MATH	GSM8K
MAPRO	80.21 ± 0.90	76.54 ± 0.67	31.52 ± 0.61	64.00 ± 0.35	34.65 ± 0.30	64.30 ± 0.59	93.48 ± 0.42
w/o demos	76.04 ± 0.90	75.22 ± 0.31	29.70 ± 0.61	62.33 ± 0.31	34.20 ± 0.35	63.87 ± 0.23	92.86 ± 0.15
Drop (%)	5.20%	1.72%	5.78%	2.61%	1.30%	0.67%	0.66%

Table 4: Ablation study results showing the performance drop when removing demonstration-guided reward. Numbers are reported with mean ± standard deviation, and relative drops are given in percentage.

plex reasoning. For LLM backbones, the results reaffirm general trends—Haiku excels in code, whereas Llama is stronger in reasoning—but also show that MAPRO adapts well across both. Notably, the optimal MAS under Llama shifted toward more sophisticated designs, suggesting that stronger reasoning models further amplify the benefits of our framework. Overall, these results validate MAPRO as both more effective and more versatile than existing methods, with significant potential for even greater gains on future LLMs.

4.3 Optimization Trajectory

We visualize the optimization trajectory of MAPRO as shown in Figure-3. MAPRO's trajectory demonstrates a more steady trend of optimization that gradually improves the validation performance towards better prompt sets, whereas we observe more fluctuations when it comes to other optimization methods, as they have a hard time capturing complicate interplays between agents. We further inspect an example of optimized prompt trajectory of an agent node in Table-2. As can be seen, the prompt evolves overtime with more precise instructions that provides task-specific insights. These insights, especially the repeatedly occurring refinements, in practice, can be ingested into knowledge base which facilitates human-in-the-loop process and bring in more reliability and robustness to the system.

4.4 Reward Model Analysis

To assess the incremental gains of the reward models, we conducted an ablation study across all tasks. As shown in Table 4, the results underscore the critical role of demonstration-guided reward, consistent with TPO (Li et al., 2025). A key insight is that the contribution of demonstrations varies across tasks, likely due to the relative simplicity of certain benchmarks such as those in the math domain. We also examined the consistency of the scoring process: under a low temperature setting, the selection procedure produced nearly identical outcomes across tasks and MAS configurations, so we omit ablation on that front. This robustness, together with the ablation results, demonstrate the efficacy of our reward model design.

4.5 Efficiency and Cost Analysis

Because absolute runtime and monetary cost vary substantially across API providers and organizational infrastructure, we adopt the number of *agent-level LLM calls* as the most fair and comparable efficiency metric. A detailed cost analysis will be included in the revised version. Below we provide a concise comparison.

- **Direct optimization.** Each iteration updates all agents once, resulting in N LLM calls.

- **TPE-style methods.** The agent update stage requires N calls. In addition, the search stage evaluates S sampled joint prompt configurations, each requiring a full multi-agent system (MAS) execution, incurring an additional $S \times N$ calls.

- **MAPRO (ours).** The update stage similarly requires N calls. The search stage relies on factored scoring, consisting of N node-level scores and $E \times K$ edge-factor scores, which can be efficiently batched. Only a single MAS execution is required per iteration, as the global configuration is obtained via MAP inference rather than repeated rollout evaluations.

Overall, an objective efficiency comparison is non-trivial. When S is small, TPE-style methods tend to converge slowly and inefficiently; when S is large, they may require more LLM calls than MAPRO. In our experiments, following standard practice, we set $S = 20$, under which TPE-style methods are only marginally faster than MAPRO. Nevertheless, as shown in Figure 3, MAPRO consistently achieves superior performance under a comparable call budget, primarily because MAP inference avoids poor local optima and enables more comprehensive exploration of the search space.

5 Conclusion

We introduced MAPRO, a principled framework that first recasts multi-agent prompt optimization as a MAP inference problem and resolves it through language-guided belief propagation and topology-aware refinement. Across diverse downstream tasks, MAPRO consistently surpasses all types of baselines, demonstrating its effectiveness and generality. Beyond strong empirical gains, MAPRO delivers a plug-and-play setting that balances accuracy with flexibility. This flexibility, together with the interpretability provided by the optimization trajectories, making MAPRO especially practical for real-world MAS deployment and improvement.

Acknowledgements

The work of Z. Zhang, C. Zhang, and Y. Ye was partially supported by the NSF under grants IIS-2533550, IIS-2321504, IIS-2217239, CNS-2426514, and CMMI-2146076, Notre Dame Strategic Framework Research Grant (2025), and Notre Dame Poverty Research Package (2025). Any expressed opinions, findings, and conclusions or recommendations are those of the authors and do not necessarily reflect the views of the sponsors.

Limitations

In this section, we discuss the limitations of our work and outline promising directions for future research. First, our study focuses exclusively on optimizing prompts for MASs while keeping the agent topology fixed. The results suggest that additional gains could be achieved through more deliberate choices of MAS topology. However, updating the topology requires reconfiguring the entire system, which is substantially more complex and resource-intensive. In many industrial applications where MAS designs are already deployed or constrained by fixed requirements, such re-establishment is impractical. This reflects a fundamental trade-off between flexibility, extensibility, efficiency, and performance. Nevertheless, extending MAPRO to jointly optimize both prompts and topologies would be an exciting avenue for future exploration. Second, while we employed LLM-based agents as reward models and demonstrated their efficacy and consistency, it would be valuable to investigate fine-tuned alternatives. In particular, approaches such as Max a Posteriori Policy Optimization (Abdolmaleki et al., 2018) offer a principled framework that could replace our current reward mechanism and integrate more seamlessly into the overall optimization process. Exploring such directions could further enhance the robustness and generality of our approach.

References

Abbas Abdolmaleki, Jost Tobias Springenberg, Yuval Tassa, Remi Munos, Nicolas Heess, and Martin Riedmiller. 2018. Maximum a posteriori policy optimisation. In *ICLR*.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on freebase from question-answer pairs. In *EMNLP*.

Chi-Min Chan, Weize Chen, Yusheng Su, Jianxuan Yu, Wei Xue, Shanghang Zhang, Jie Fu, and Zhiyuan Liu. 2024. Chateval: Towards better llm-based evaluators through multi-agent debate. In *ICLR*.

Jingchang Chen, Hongxuan Tang, Zheng Chu, Qianglong Chen, Zekun Wang, Ming Liu, and Bing Qin. 2024a. Divide-and-conquer meets consensus: Unleashing the power of functions in code generation. In *NeurIPS*.

Yongchao Chen, Jacob Arkin, Yilun Hao, Yang Zhang, Nicholas Roy, and Chuchu Fan. 2024b. Prompt optimization in multi-step tasks (promst): Integrating human feedback and heuristic-based sampling. In *EMNLP*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Wendi Cui, Jiaxin Zhang, Zhuohang Li, Hao Sun, Damien Lopez, Kamalika Das, Bradley A Malin, and Sricharan Kumar. 2025. Automatic prompt optimization via heuristic search: A survey. *arXiv*.

Nicola Dainese, Matteo Merler, Minttu Alakuijala, and Pekka Marttinen. 2024. Generating code world models with large language models guided by monte carlo tree search. In *NeurIPS*.

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. 2025. Codescore: Evaluating code generation by learning code execution. *ACM Transactions on Software Engineering and Methodology*.

Xidong Feng, Bo Liu, Yan Song, Haotian Fu, Ziyu Wan, Girish A Koushik, Zhiyuan Hu, Mengyue Yang, Ying Wen, and Jun Wang. 2024. Natural language reinforcement learning. *arXiv preprint arXiv:2411.14251*.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2024. Promptbreeder: self-referential self-improvement via prompt evolution. In *ICML*.

Lin Ge, Hengrui Cai, Runzhe Wan, Yang Xu, and Rui Song. 2025. A review of causal decision making. *arXiv preprint arXiv:2502.16156*.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. 2024a. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *ICLR*.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. 2025. Evoprompt: Connecting llms with evolutionary algorithms yields powerful prompt optimizers. *arXiv*.

Siyuan Guo, Cheng Deng, Ying Wen, Hechang Chen, Yi Chang, and Jun Wang. 2024b. Ds-agent: automated data science by empowering large language models with case-based reasoning. In *ICML*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. In *NeuralIPS*.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. 2024a. Metagpt: Meta programming for a multi-agent collaborative framework. In *ICLR*.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. 2024b. Metagpt: Meta programming for a multi-agent collaborative framework. In *ICLR*.

Shengran Hu, Cong Lu, and Jeff Clune. 2025. Automated design of agentic systems. In *ICLR*.

Haitao Jiang, Lin Ge, Yuhe Gao, Jianian Wang, and Rui Song. 2024. Llm4causal: Democratized causal tools for everyone via large language model. In *CoLM*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *ICLR*.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, Heather Miller, et al. 2024. Dspy: Compiling declarative language model calls into state-of-the-art pipelines. In *ICLR*.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *NeurIPS*.

Chao Lei, Yanchuan Chang, Nir Lipovetzky, and Krista A Ehinger. 2025. Planning-driven programming: A large language model programming workflow. *ACL*.

Manling Li, Shiyu Zhao, Qineng Wang, Kangrui Wang, Yu Zhou, Sanjana Srivastava, Cem Gokmen, Tony Lee, Li Erran Li, Ruohan Zhang, et al. 2024. Embodied agent interface: benchmarking llms for embodied decision making. In *NeuralIPS*.

Yafu Li, Xuyang Hu, Xiaoye Qu, Linjie Li, and Yu Cheng. 2025. Test-time preference optimization: On-the-fly alignment via iterative textual feedback. In *ICML*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *NeuralIPS*.

Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Shieh, and Wenmeng Zhou. 2025a. Codexgraph: Bridging large language models and code repositories via code graph databases. *NAACL*.

Yexiang Liu, Jie Cao, Zekun Li, Ran He, and Tieniu Tan. 2025b. Breaking mental set to improve reasoning through diverse multi-agent debate. In *ICLR*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *NeurIPS*.

Krista Opsahl-Ong, Michael Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. 2024. Optimizing instructions and demonstrations for multi-stage language model programs. In *EMNLP*.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *NeuralIPS*.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2025. Repograph: Enhancing ai software engineering with repository-level code graph. *ICLR*.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 2024. Chatdev: Communicative agents for software development. In *ACL*.

Chen Qian, Zihao Xie, YiFei Wang, Wei Liu, Kunlun Zhu, Hanchen Xia, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, et al. 2025. Scaling large language model-based multi-agent collaboration. In *ICLR*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *NeurIPS*.

Aditi Singh, Abul Ehtesham, Saket Kumar, and Tala Talaei Khoei. 2025. Agentic retrieval-augmented generation: A survey on agentic rag. *arXiv*.

Feifan Song, Yuxuan Fan, Xin Zhang, Peiyi Wang, and Houfeng Wang. 2025. Instantly learning preference alignment via in-context dpo. In *NAACL*.

Adam Trischler, Tong Wang, Xingdi Yuan, Justin Harris, Alessandro Sordoni, Philip Bachman, and Kaheer Suleman. 2016. Newsqa: A machine comprehension dataset. *arXiv*.

Junlin Wang, WANG Jue, Ben Athiwaratkun, Ce Zhang, and James Zou. 2025. Mixture-of-agents enhances large language model capabilities. In *ICLR*.

Ruochen Wang, Sohyun An, Minhao Cheng, Tianyi Zhou, Sung Ju Hwang, and Cho-Jui Hsieh. 2024a. One prompt is not enough: automated construction of a mixture-of-expert prompts. In *ICML*.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024b. Executable code actions elicit better llm agents. In *ICML*.

Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric Xing, and Zhiting Hu. 2024c. Promptagent: Strategic planning with language models enables expert-level prompt optimization. In *ICLR*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *NeuralIPS*.

Zhaoxuan Wu, Xiaoqiang Lin, Zhongxiang Dai, Wenyang Hu, Yao Shu, See-Kiong Ng, Patrick Jaillet, and Bryan Kian Hsiang Low. 2024. Prompt optimization with ease? efficient ordering-aware automated selection of exemplars. In *NeurIPS*.

Jinyu Xiang, Jiayi Zhang, Zhaoyang Yu, Fengwei Teng, Jinhao Tu, Xinbing Liang, Sirui Hong, Chenglin Wu, and Yuyu Luo. 2025. Self-supervised prompt optimization. *arXiv*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *ICLR*.

Yanfang Ye, Zheyuan Zhang, Tianyi Ma, Zehong Wang, Yiyang Li, Shifu Hou, Weixiang Sun, Kaiwen Shi, Yijun Ma, Wei Song, et al. 2025. Llms4all: A review of large language models across academic disciplines. *arXiv preprint arXiv:2509.19580*.

Zhengqing Yuan, Yixin Liu, Yihan Cao, Weixiang Sun, Haolong Jia, Ruoxi Chen, Zhaoxu Li, Bin Lin, Li Yuan, Lifang He, et al. 2024. Mora: Enabling generalist video generation via a multi-agent framework. *arXiv preprint arXiv:2403.13248*.

Guibin Zhang, Yanwei Yue, Xiangguo Sun, Guancheng Wan, Miao Yu, Junfeng Fang, Kun Wang, Tianlong Chen, and Dawei Cheng. 2025a. G-designer: Architecting multi-agent communication topologies via graph neural networks. *ICML*.

Zheyuan Zhang, Kaiwen Shi, Zhengqing Yuan, Zehong Wang, Tianyi Ma, Keerthiram Murugesan, Vincent Galassi, Chuxu Zhang, and Yanfang Ye. 2025b. Agentrouter: A knowledge-graph-guided llm router for collaborative multi-agent question answering. *arXiv preprint arXiv:2510.05445*.

Kunhao Zheng, Juliette Decugis, Jonas Gehring, Taco Cohen, Gabriel Synnaeve, et al. 2025. What makes large language models reason in (multi-turn) code generation? In *The Thirteenth International Conference on Learning Representations*.

Han Zhou, Xingchen Wan, Lev Proleev, Diana Mincu, Jilin Chen, Katherine A Heller, and Subhrajit Roy. 2024. Batch calibration: Rethinking calibration for in-context learning and prompt engineering. In *ICLR*.

Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan Ö Arık. 2025. Multi-agent design: Optimizing agents with better prompts and topologies. *arXiv*.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. 2024. Gptswarm: language agents as optimizable graphs. In *ICML*.

A Related Work

A.1 Prompt Optimization for MAS

Recent progress in large language models (LLMs) has enabled multi-agent systems (MAS), in which cooperating agents consistently outperform single agents on demanding reasoning and software-engineering tasks (Zhang et al., 2025b; Ye et al., 2025; Wang et al., 2024b). This performance, however, depends on labor-intensive prompt engineering: each agent needs carefully crafted role instructions, and this effort grows rapidly as the number of agents increases, because aligning coordination between them becomes increasingly complex. To ease this burden, many studies now frame prompt design as an optimization problem and use heuristic search algorithms to explore and refine prompts with minimal human effort and oversight.

Prompt optimization in LLMs generally falls into two main categories: soft-prompt tuning in continuous space and discrete prompt optimization in text space. Soft tuning supports gradient-based updates but sacrifices transparency and portability, as its learned vectors are opaque, model-specific, and require gradient access that most black-box APIs do not provide (Cui et al., 2025). To work around this, researchers approximate gradients with LLM feedback and develop gradient-like strategies suited for non-differentiable settings. For example, some works apply beam search for step-wise refinement (Chen et al., 2024b; Wang et al., 2024c), while others explore alternative optimization strategies, such as evolutionary algorithms (Guo et al., 2025; Fernando et al., 2024) and other heuristic algorithms (Opsahl-Ong et al., 2024; Li et al., 2025), to adapt prompts iteratively. Another related line of work focuses on prompt selection, searching a pool of variants to pick the best one (Wu et al., 2024; Wang et al., 2024a; Song et al., 2025).

However, most existing methods target a single agent, while prompt optimization for MAS as a whole remains under-explored. Among the few efforts in this area, Mass (Zhou et al., 2025) warms each agent's role prompt, prunes the interaction graph, and then jointly fine-tunes all prompts, showing that layered optimization boosts group performance. GPTSwarm (Zhuge et al., 2024) treats agents as a graph and updates node-level prompts and edge connections together, letting prompts co-evolve with coordination patterns. We argue that the importance of proper prompt design has been significantly under-studied in these prior works.

Specifically, current approaches still overlook two key issues: even minor lexical edits upstream can shift the distributions seen by downstream agents, and, to the best of our knowledge, none of the methods searches for a globally optimal set of prompts for the full system.

A.2 LLM Agents for Code Generation Tasks

Code generation has emerged as a core application for large language model (LLM) agents because it links natural-language reasoning with concrete, testable outputs and promises to automate sizable portions of software engineering and data-science workflows. Early studies tackled this task with single-agent or minimally interactive pipelines—such as Self-Refine (Madaan et al., 2023), Reflexion (Shinn et al., 2023), and CoT-Zero (Kojima et al., 2022)—that plan, execute, and iteratively repair their own code until unit tests pass. These works showed that even simple agent interactions can improve reliability when the agent can inspect failures and revise its output, a process that loosely aligns with causal reasoning (Ge et al., 2025; Jiang et al., 2024): the model infers potential sources of error and adjusts its generation in response.

As multi-agent systems advance, the trend has shifted toward complex MAS with carefully designed role-play interactions. Frameworks like CodeAct (Wang et al., 2024b), MetaGPT (Hong et al., 2024b), and ChatDev (Qian et al., 2024) assign specialized roles—planner, coder, tester, reviewer—and let agents converse in plain language, mimicking real software teams. These orchestrated exchanges boost division of labor and help solve coding problems that demand sophisticated reasoning, though they impose substantial overhead in prompt design. Building on these frameworks, recent work explores several directions. Some studies enhance individual modules through richer planning (Lei et al., 2025; Chen et al., 2024a), stronger verification (Dainese et al., 2024; Zheng et al., 2025), or improved knowledge bases (Ouyang et al., 2025; Liu et al., 2025a). Others introduce supervised signals into the framework, such as reinforcement learning (Feng et al., 2024), to guide agent behavior.

However, across all phases, prompt design remains a bottleneck: each role prompt must be carefully crafted, and even minor edits can ripple through the workflow. Consequently, a growing body of research now investigates automatic prompt optimization (Zhang et al., 2025a; Zhou et al., 2025) to unlock more reliable and generalizable agent-collaboration schemes for real-world coding tasks.

B Implementation Details

B.1 Benchmarks

HumanEval-ET (Dong et al., 2025) is an extended benchmark for evaluating code generation. It builds upon the original HumanEval dataset by introducing more challenging variations and refined evaluation protocols, particularly emphasizing error tolerance and execution-based correctness. The dataset is specifically designed to better capture the robustness of large language models (LLMs) under real-world coding scenarios, where multiple correct implementations may exist and minor deviations from reference solutions should not necessarily be penalized. By incorporating these refinements, HumanEval-ET provides a more reliable and nuanced measure of code generation quality. Since this dataset doesn't provide a train-test split, we used the first 100 records for optimization and the rest 64 for zero-shot left-out testing.

MBPP-Plus (Liu et al., 2023) extends the "Mostly Basic Python Problems" (MBPP) dataset into a larger and more diverse collection. While MBPP was originally created to evaluate basic programming competency using short Python functions, MBPP-Plus expands both the scale and variety of tasks to cover more intricate programming constructs, edge cases, and multi-step logic. This augmentation addresses the limitations of the original dataset by providing a broader set of problems that better reflect practical coding challenges, thereby serving as a more comprehensive benchmark for evaluating code generation models.

CodeContest (Li et al., 2022) is a benchmark derived from real competitive programming problems, representing a significant increase in difficulty compared to synthetic or basic coding datasets. It contains tasks sampled from programming competitions, where problems are designed to require algorithmic reasoning, data structure manipulation, and efficiency considerations. The inclusion of strict input–output constraints and hidden test cases makes CodeContest a rigorous benchmark that challenges LLMs to go beyond template-based solutions and demonstrate genuine problem-solving ability. Given the large volume of this dataset's training set, we sample the same records as the test set from training for optimization.

NewsQA (Trischler et al., 2016) is a large-scale question answering dataset constructed from CNN news articles. It consists of over 100,000 human-generated questions paired with answers derived from corresponding news passages. Unlike earlier QA datasets that focus on simple fact extraction, NewsQA emphasizes reasoning, inference, and synthesis across multiple sentences within an article. Its design introduces ambiguity, unanswerable questions, and multi-sentence reasoning, making it a challenging benchmark for evaluating reading comprehension and open-domain question answering systems. Given the large volume of this dataset, we sample the first 500 records to use as optimization and left-out testing.

WebQuestions (Berant et al., 2013) is a benchmark dataset for semantic parsing and knowledge-base question answering. It contains around 6,000 natural language questions paired with answers sourced from Freebase, covering a diverse range of topics. The dataset is notable for requiring models to bridge the gap between natural language queries and structured knowledge graph representations, thereby testing a system's ability to perform entity linking, relation extraction, and logical reasoning. As one of the earliest large-scale QA datasets grounded in knowledge bases, WebQuestions has been widely adopted as a standard benchmark for semantic parsing and open-domain QA research. Given the large volume of this dataset, we sample the first 500 records to use as optimization and left-out testing.

MATH (Hendrycks et al., 2021) is a dataset specifically designed to evaluate advanced mathematical reasoning in LLMs. It contains approximately 12,000 competition-style problems, ranging from high school mathematics to Olympiad-level challenges, with step-by-step solutions provided. Unlike arithmetic-focused datasets, MATH covers a broad spectrum of topics including algebra, geometry, number theory, and calculus, requiring multi-step reasoning and symbolic manipulation. Its complexity makes it one of the most rigorous benchmarks for assessing the capacity of LLMs to handle formal reasoning and mathematical problem-solving. Given the large volume of this dataset's training set, we sample the same records as the test set from training for optimization.

GSM8K (Cobbe et al., 2021) (Grade School Math 8K) is a benchmark comprising 8.5k carefully crafted grade-school-level math word problems. Each problem is designed to require multi-step reasoning with arithmetic operations, testing a model's ability to parse natural language descriptions, translate them into formal reasoning steps, and compute the correct answer. The dataset emphasizes chain-of-thought reasoning and has become a standard testbed for evaluating LLMs' ability to perform reliable symbolic reasoning in relatively simple but compositional tasks. Its structured design and moderate difficulty level make GSM8K complementary to more advanced datasets like MATH. Given the large volume of this dataset's training set, we sample the same records as the test set from training for optimization.

B.2 Baselines

Chain-of-Thought (CoT) (Wei et al., 2022) is a prompting paradigm that encourages large language models (LLMs) to generate intermediate reasoning steps before arriving at final answers. Unlike direct-answer prompting, CoT exposes the model's latent reasoning process, which has been shown to substantially improve performance on tasks requiring multi-step deduction such as arithmetic, commonsense inference, and symbolic reasoning. The introduction of CoT has established a new standard for eliciting reasoning from LLMs, making it a fundamental baseline in subsequent research. Its effectiveness also highlights a broader principle: structured prompting can significantly extend the reasoning capability of LLMs without the need for additional training.

ReAct (Yao et al., 2023) builds upon CoT by integrating reasoning with acting. Specifically, ReAct enables agents to interleave chain-of-thought reasoning with concrete actions, such as querying external knowledge sources, interacting with environments, or calling tools. This synergy allows models to dynamically refine their reasoning based on external feedback, thereby reducing hallucinations and improving factual grounding. ReAct has been validated across diverse tasks including knowledge-intensive QA, fact verification, and embodied agent settings, where its reasoning-and-acting paradigm consistently outperforms reasoning-only or acting-only strategies. As a baseline, ReAct represents an important step toward interactive and tool-augmented LLM systems.

EvoPrompt (Guo et al., 2024a) frames prompt optimization as an evolutionary search process, where a population of prompts is iteratively mutated and recombined to generate stronger candidates. The method relies on large language models themselves as operators for variation, while selection mechanisms ensure gradual improvement. This makes EvoPrompt effective for black-box single-agent prompt optimization. However, its design remains confined to evolving isolated prompts, and it does not extend naturally to multi-agent settings where inter-agent coordination and topology play central roles.

PromptBreeder (Fernando et al., 2024) extends evolutionary prompt optimization by introducing self-referential mutation. In this framework, not only task-prompts but also the mutation-prompts that generate them are evolved, enabling the system to adapt its own optimization strategy over time. This self-referential design yields a flexible and automated process for refining prompts in single-agent contexts. Nevertheless, PromptBreeder is inherently tailored to optimizing individual prompts and does not address the complexities of scaling to multi-agent systems.

Chain (Shinn et al., 2023) represents the simplest combination topology of MAS. In our study, it combines the reasoning-and-acting paradigm of ReAct with a self-reflection module. After completing a task, the agent revisits its reasoning trajectory, identifies mistakes, and integrates corrective feedback into subsequent attempts. By incorporating the simple reflection module into the loop, Chain improves both robustness and sample efficiency. In our experiments, we adopt this variant as a baseline to capture the benefits of the effectiveness of simple MAS, compared with other MAS choices.

DMAD (Liu et al., 2025b) (Diverse Multi-Agent Debate) is a recent state-of-the-art framework designed to overcome the inherent limitations of multi-agent debate (MAD). Traditional MAD setups often fall prey to a mental set, where agents—even if assigned different personas—rely on similar reasoning strategies, limiting their ability to explore alternative solutions. DMAD explicitly addresses this by requiring each agent to employ a distinct reasoning method (e.g., Chain-of-Thought, Step-Back Prompting, Program-of-Thought), thereby fostering genuine diversity in problem-solving. This represents an intermediate-complexity MAS topology that balances complexity and expressiveness. Given its robustness and state-of-the-art results, we consider DMAD an essential MAS base structure for evaluating MAS-level optimization.

ChatEval (Chan et al., 2024) is a multi-agent evaluation framework that leverages structured dialogue among diverse LLM agents to produce more reliable judgments. In our study, we adopt the *Simultaneous-Talk-with-Summarizer* variant, which we call SWARM, where agents contribute in parallel and a summarizer condenses their discussion into a concise shared history. In this setting, each agent interacts with each other in a dense format, making this baseline a typical and representative topology type, as it emphasizes on the richness of multi-agent discussion with strong reasoning and expressiveness.

GPTSwarm (Zhuge et al., 2024) frames language agents as computational graphs, where each node corresponds to an operation such as an LLM query, and edges capture the flow of information across agents. This framework is intended for optimizing both topology and prompts. To enable a fair comparison, we focus solely on the prompt optimization parts of this work. Within this framework, **Direct Optimization** is employed to refine the prompts associated with each node individually, using input–output histories and iterative updates to improve local performance. This strategy allows each operation to self-improve in isolation, but it treats prompts largely as independent units and does not account for the interdependencies across the wider agent graph. In contrast, our MAPRO framework explicitly models prompt optimization as a joint inference problem over the entire MAS topology, propagating credit and dependencies across nodes and edges. Thus, while GPTSwarm provides a strong formulation of node-level direct optimization, our approach generalizes this idea to coordinated optimization across multi-agent systems, addressing the limitations of local-only updates

TPE Optimization (Zhou et al., 2025; Opsahl-Ong et al., 2024) applies a Tree-structured Parzen Estimator (TPE)–based Bayesian search strategy to optimize prompts in multi-agent systems. In these frameworks, prompts (instructions and demonstrations) are treated as discrete parameters, and TPE is used to model the joint contribution of different parameter settings to downstream performance. This surrogate-based approach efficiently explores the search space by prioritizing promising configurations from past evaluations. Unlike **Direct Optimization**, which treats node prompts independently, TPE can partially capture dependencies between variables through its probabilistic modeling. However, TPE optimization still operates over a fixed pool of candidate proposals, limiting its ability to propagate credit across agents or adapt proposals dynamically. In contrast, our MAPRO framework formulates prompt optimization as a joint inference problem across the entire MAS topology, explicitly propagating dependencies and credit signals across nodes and edges. This enables coordinated optimization beyond the local or surrogate-based updates employed by TPE methods, addressing their limitations in capturing the full structure of multi-agent interactions.

B.3 Training Protocol

We limit the number of preference demonstrations to 3 and candidates to 5. We limit the agent number smaller than 10. We set model temperature at 0.2, maximum output tokens at 2048. We implement the same LLM backbone as both evaluator and executors in all phases. The optimized MAS is reported on the held-out test set over three runs, while other baselines over five runs. Given our mission to optimize the prompts, we didn't spend too much effort on prompt engineering, which mimics the real-life scenarios where a general prompt is adopted to a specific downstream tasks. The specific prompt designs can be seen in Appendix-D.

C Proof

C.1 Proof of MAP Equivalence

By Bayes' rule, the classic MAP estimate chooses the hypothesis P that maximizes the posterior:

$$\begin{aligned}\widehat{P}_{\text{MAP}} &\in \arg\max_{P}\ \Pr(P \mid S) \\ &= \arg\max_{P}\ \Pr(S \mid P)\,\pi(P),\end{aligned} \tag{12}$$

where S is the observed event and $\pi(P)$ is the prior on P. This way MAP finds the most probable explanation (the most likely hidden variable assignment) given what one observed.

In our case, $P = (p_1, \ldots, p_N) \in P_1 \times \cdots \times P_N$ is a joint prompt assignment and S denote the event that the system succeeds on the batch. By construction of the node/edge success scores $g(\cdot), g(\cdot, \cdot)$,

$$\begin{aligned}\Pr(S \mid P) &= \prod_{i=1}^{N} \Pr(X_i{=}1 \mid P) \prod_{(i,j)\in\mathcal{E}} \Pr(Y_{ij}{=}1 \mid P) \\ &= \prod_{i=1}^{N} g(p_i) \prod_{(i,j)\in\mathcal{E}} g(p_i, p_j) \\ &=: \mathcal{T}(P).\end{aligned} \tag{13}$$

Since we do not assume one prompt set is inherently better than another (we have no prior knowledge). The most neutral choice is to use a uniform prior, and under a uniform prior, every $P \in P_1 \times \cdots \times P_N$ is assigned the same positive probability. Thus $\pi(P) = c$ for some constant $c > 0$ independent of P. Since multiplying by a constant does not affect an $\arg\max$, we have

$$\arg\max_{P} f(P)\, c = \arg\max_{P} f(P).$$

Therefore, given $\mathcal{T}(P) = \Pr(S \mid P)$,

$$\begin{aligned}\arg\max_{P} \Pr(P \mid S) &= \arg\max_{P} \Pr(S \mid P)\,\pi(P) \\ &= \arg\max_{P} \Pr(S \mid P)\, c \\ &= \arg\max_{P} \Pr(S \mid P) \\ &= \arg\max_{P} \mathcal{T}(P).\end{aligned} \tag{14}$$

Thus, maximizing the Joint Quality Score is exactly a MAP estimate of P.

C.2 Proof of Junction Tree MAP

Max-product belief propagation (MPBP) is guaranteed to compute the exact MAP assignment only on tree-structured factor graphs. For a DAG $G = (\mathcal{V}, \mathcal{E})$, the factorization ($\mathcal{T}(P)$) generally induces cycles, since a node j with multiple parents couples the variables $\{p_i : (i,j) \in \mathcal{E}\}$ together. Formally, one first moralizes and triangulates the DAG to ensure a chordal structure admitting a junction tree.

The junction-tree construction converts this DAG factorization into an equivalent tree-structured form. The procedure groups variables into clusters $C \subseteq \mathcal{V}$, each associated with a potential ψ_C defined as

$$\psi_C(P_C) := \prod_{i\in C} g(p_i) \prod_{\substack{(i,j)\in\mathcal{E} \\ \{i,j\}\subseteq C}} g(p_i, p_j), \tag{15}$$

where $P_C = \{p_i : i \in C\}$. In words, every factor is assigned to exactly one cluster that contains its variables. Clusters are arranged in a tree $\mathcal{T}_{\text{JT}}$ satisfying the *running intersection property*: if a variable p_i appears in two clusters C_1, C_2, then it appears in every cluster on the unique path between C_1 and C_2 in $\mathcal{T}_{\text{JT}}$.

The resulting representation is an exact refactorization:

$$\mathcal{T}(P) = \prod_{C\in\mathcal{C}} \psi_C(P_C) \,/\, \prod_{s\in\mathcal{S}} \psi_s(P_s), \tag{16}$$

where $\mathcal{S}$ denotes the separator sets (intersections of adjacent clusters). Here each separator potential ψ_s is defined as the product of factors assigned to s, ensuring no double counting. The division by separators ensures that no factor is double-counted and that the product reproduces exactly $\mathcal{T}(P)$. Since this is equivalent to the original joint score $\mathcal{T}(P)$ but expressed on a tree-structured factor graph, applying MPBP to $\{\psi_C\}$ on the junction tree yields

$$\arg\max_P \mathcal{T}(P) = \arg\max_P \frac{\prod_{C\in\mathcal{C}} \psi_C(P_C)}{\prod_{s\in\mathcal{S}} \psi_S(P_s)}. \tag{17}$$

which recovers the exact MAP assignment of P. Hence, the junction-tree transformation converts a general DAG into a tree-structured model where MPBP can be applied directly and exactly. This ratio form follows from the junction-tree theorem, which guarantees that clique potentials multiplied and corrected by separator terms reproduce the exact joint distribution.

C.3 Proof of MAP Global Optimality

Lemma (optimal-subtree property). Assume the reward factorization is finite and has no negative factors, for any edge $i \to j$ and any p_j, $m_{i\to j}(p_j)$ equals the maximum of the product of factors contained in the subtree rooted at i, conditioned on p_j.

Proof. Assume i is a leaf agent node, (6) reduces to $\max_{p_i} g(p_i)g(p_i, p_j)$, the best score of the leaf edge given p_j. Assume the claim holds for all children $k \in \mathit{Child}(i)$. Then the product inside (6) equals, for each fixed p_i, the optimal contributions of all child sub-trees consistent with p_i; maximizing over p_i yields the optimal value of the entire subtree at i given p_j.

Theorem (global MAP optimality). Let $p_r^* \in \arg\max_{p_r} \beta_r(p_r)$. Then there exists an assignment $P^\star$ obtained by the standard downward backtracking that satisfies $P^* \in \arg\max_P \mathcal{T}(P)$,

since the root collects optimal contributions from all disjoint subtrees. Thus

$$\max_{p_r} \beta_r(p_r) = \max_P \mathcal{T}(P).$$

During the upward pass, for every edge $i \to j$ and parent value p_j, the maximizer(s) achieving (6) define a witness choice $p_i^\star(p_j)$. Starting from $p_r^* \in \arg\max \beta_r(p_r)$ and recursing $p_i^\star\left(p_j^*\right)$ along edges away from r yields a full assignment P^* that realizes the global maximum (ties broken arbitrarily).

Remark (junction tree). In the clique/sepset form, replace nodes i by cliques C, parent j by neighbor D, p_i by P_C, and g by ψ; messages are

$$m_{CD}(P_{S_{CD}}) = \max_{P_C \setminus S_{CD}} \left[\psi_C(P_C) \prod_{B \in \mathrm{nb}(C)\setminus\{D\}} m_{BC}(P_{S_{BC}})\right],$$

and the same induction establishes exact MAP for the original DAG via the established equivalence.

C.4 Time Complexity Analysis

MPBP on a tree. Let $N = |\mathcal{V}|$ be the number of agents, $E = |\mathcal{E}|$ the number of edges, and $K = \max_i |P_i|$ the maximum size of any agent's prompt pool. On a tree-structured factor graph, each edge passes one message in each direction. Updating a single message requires comparing all prompt pairs (p_i, p_j), which costs $O(K^2)$. Since there are $O(E)$ such messages in total, the overall complexity is

$$O(EK^2).$$

Junction-Tree MAP (general DAG). For a DAG, we convert the graph into a junction tree whose clique set is denoted $\mathcal{C}$. Let w be the induced treewidth, i.e. the size of the largest clique minus one. Each message update involves marginalizing/maximizing over a clique table of size $O(K^{w+1})$, and there are $O(|\mathcal{C}|)$ such cliques (at most linear in N). Thus the complexity is

$$O(|\mathcal{C}|\, K^{w+1}),$$

with storage requirements of the same order.

The treewidth w reflects how many agents must be grouped into a single clique to remove cycles. For instance, if two agents both depend on the same parent, they may be merged into a summary clique of size three, so $w = 2$. If a node has three parents, then a clique containing all four variables may be needed, giving $w = 3$. In general, sparse MAS graphs usually have small w (often 2 or 3), so the exponential factor K^{w+1} remains modest. This means that in practical multi-agent settings, where each agent only interacts with a few neighbors, junction-tree MAP is efficient and scales nearly linearly with N once w is bounded.

In summary, we control both selection and update phase in polynomial time complexity and it scales well the increase of the density of interactions as well as the size of candidate pools, which emphasize the scalability and efficiency of our MAPRO framework.

D Prompt Designs

In this section, we provide the prompts for all base agents, and all helper-agent prompts (judge, variation generator, critic, etc.). These components are essential for interpretability and reproducibility.

It's worth noting, to evaluate baselines prompt optimization in multi-agent systems, which is intrinsically a plug-and-play setting, we adopt existing MAS designs from prior work (Swarm, DMAD). Their agent counts, roles, and base prompts originate directly from the corresponding papers, and therefore were not listed here as methodological contributions.

Node-Level Reward Model (Header + Prefix)

node_header:
You are a *reward model* for evaluating the competence, clarity of candidate **role prompts**.
Based on the input, output and prefernece examples,
you should first rank the candidate prompts with the good and bad examples,
Then you will give each a distinct two-decimal quality score between (0.00, 1.00) based on the standard and alignment with the good examples.
You should be severely harsh and the score difference should be ranged from 0.4 - 0.8 and each differs more than 0.05 with each other.
Finally, return exactly a score each line corresponding to the **prompt's original position**. (Not the sorted score)
Note that your output should contain only the numeric scores (e.g., 0.62). Nothing else.

agent_reward_prefix:
You are an evaluation LLM. Given {input} and the agent's response {output}, rate how well the response accomplishes the agent's role on a scale 0–1 (higher is better).Use the preference demonstrations below as reference.Return ONLY the floating-point score.

=== Preference Demonstrations ===
{demo}
=== End Demonstrations ===

Edge-Level Reward Model (Header + Prefix)

edge_header:
You are a *reward model* for assessing **communication quality** from
an upstream agent to a downstream agent. Consider information completeness, format,
clarity, and alignment with demonstrations.
Based on the input, output and prefernece examples,
you should first rank the candidate prompts with the good and bad examples,
Then you will give each a distinct two-decimal quality score between (0.00, 1.00) based on the standard and alignment with the good examples.
You should be severely harsh and the score difference should be ranged from 0.4 - 0.8 and each differs more than 0.05 with each other.
Finally, return exactly a score each line corresponding to the **prompt's original position**. (Not the sorted score)
Note that your output should contain only the numeric scores (e.g., 0.62). Nothing else.

edge_reward_prefix:
You are an evaluation LLM. Judge whether a message produced by agent {i} helps agent {j} perform its next step. Rate on a 0–1 scale. Use the demonstrations for guidance. Return ONLY the floating-point score.

=== Preference Demonstrations ===
{demo}
=== End Demonstrations ===

Figure 4: Unified Reward Modeling Prompts for MAPRO: node-level (left) and edge-level (right), merging each module's header and reward prefix verbatim.

Feedback and Mutation Strategy Prompts

global_feedback_sys:
You are an experienced prompt engineer and failure-analysis specialist.
Given multiple examples of runtime *error messages* produced by the given LLM-generated code,
identify the three most recurring but easy to solve root-cause patterns or missing constraints **in the prompts** that lead to the errors. Produce a short **specific and actionable** list of fix suggestions an author can apply.
Note 1: Output each fix as a bullet starting with numbers. Do NOT quote full stack traces; mention key function names only if essential.
Note 2: You should focus on the pragmatism and cleaniness of code rather than if it's easy to read, for example, if the a module doesn't have package ‘List‘, instead of asking to properly import the package, you should emphasize it should write code without any type hints or annotations.

local_feedback_sys:
You are a experienced prompt engineer and failure-analysis specialist. You are given:
1) The global overall feedback list that the system is currently facing.
2) Blame statements from downstream agents suggesting how the current module can be improved (may be empty).
3) The prompt this module is currently using.
Based on the roles of the current module, your task is to generate a *local feedback* list, focusing on give specific, actionable fix suggestions specifically for this current module to take to avoid downstream errors and satisfy the overall fix suggestions. Each line starts with ‘•’.

mutation_strategy_sys:
You are a experienced prompt engineer and failure-analysis specialist.
You are given the original <prompt> of a module plus two feedback blocks:
One overall fix feedback suggesting the errors the system currently experience and one optional local feedback suggesting what this current modules can focus on to improve to benefit the system.
Your task is to modify, improve, and explode the original prompt by outputing exactly {n} JSON strings as prompt variations with specific and detailed improvement.
Note:
1) You should focus on the pragmatism and cleaniness of the prompts (You shouldn't acutally write any code), so **always emphasize** the code should be executable, wrapped in one function, without any type hints or annotations, and named as solution if no other names are provided.
2) You are only allowed to make relatively small edits. You must choose exactly one action item in the following: a) adding one sentence from the feedback. b) replacing one senetence from the feedback to existing edits. c) Re-organize, rewrite or clean the current prompt to make it logically consistent. d) delete one redundant sentence in the current prompt.
3) You should ALWAYS respond with ONLY the VALID JSON array – You should return No headings, no prose such as </prompt>, no markdown fences such as “‘, no trailing commas, no escape codes, or unclosed parenthesis. Each string must be valid UTF-8. Escape all newlines as \n. No raw newlines inside JSON strings. Example (node, n = 2): ["Prompt variant 1","Prompt variant 2"].

Figure 5: Feedback system prompts in MAPRO (for coding tasks): global feedback, local feedback, and mutation strategy.

Variation Prompt

variation:
You are a prompt-engineering assistant.
The user will give you an original prompt TEMPLATE inside <prompt></prompt>.
Produce {n} diverse textual prompt variants (NOT solution, but the prompts) that keep the same intent but differ in wording, ordering, or tone. Note that you should generate the prompt for the agent not generate solution.
Don't write code here and Return **only** a JSON array of strings.
Respond on a single line only. Do not emit any raw line breaks.

Negative Variation Prompt

neg_variation:
You are a prompt-mutation helper.
The user will give you a JSON object with:
good_examples : list[str] # 3 GOOD prompt templates (node) *or* 3 GOOD upstream-downstream pairs
mode : "node"|"edge" # mutation type
n : int # number of BAD variants requested
Produce exactly {n} sligthly BAD variants:
• For "node": each string could omit some key instructions, introduce contradictions, or add irrelevant text that reduces agent quality.
• For "edge": each string code be a JSON array ["bad_upstream", "good_downstream"] where bad_upstream makes the pair incompatible.
• Note that your generation should be obviously worse than good examples, but not too absurd or entirely off the topic.
Remember, Return nothing except one valid JSON array.
- For mode = "node" → ["str", "str", . . .]
- For mode = "edge" → [["str","str"], ["str","str"], . . .]
You should ALWAYS respond with ONLY the VALID JSON array – You should return No headings, no prose such as </prompt>, no markdown fences such as “‘, no trailing commas, no escape codes, or unclosed parenthesis.
Each string must be valid UTF-8. Escape all newlines as \n. No raw newlines inside JSON strings.

Figure 6: Initialization prompts in MAPRO: *variation* (left) for diverse positive variants and *neg_variation* (right) for intentionally degraded variants.

Coding Prompts and Notes

raw:
You are a reasoning agent and coding expert. Solve the task by outputting **only executable Python code** as a **single function**. Do not print any prose, comments, or markdown fences. If preprocessing or postprocessing is needed to conform to the specified input/output format, perform it *inside* the function. Follow all constraints in **note** exactly.

cot:
You are a reasoning agent and coding expert. First reason step by step *silently* (do not print thoughts or a plan). Then output **only executable Python code** as a **single function** that solves the task with the correct input/output format. Do not include comments, explanations, tags, or markdown fences. Follow **note** exactly.

react:
You are a reasoning agent. First analyze the problem and form a plan *silently* (do not print it). Then use that plan to produce the final answer as **code only** — a **single Python function** with no comments, prose, or markdown fences. If `FEEDBACK` is provided, revise the code *only* when the feedback is correct and improves conformance to the task or **note**; otherwise keep the best previous solution. Follow **note** exactly.

reflect:
You are a coding **critic**. Be conservative — revise only if there are concrete mistakes. Evaluate the submitted answer against:
1) It is **only** executable Python code with **no** markdown, tags, or comments, and has no obvious syntax errors.
2) It implements any required **preprocessing/postprocessing** so the input and output formats are correct.
3) It satisfies all constraints in **note**.
If everything is correct, reply **exactly**: `ACCEPT`
Otherwise, reply **exactly**: `REVISE: <concise, actionable fixes required>`
Do not include code blocks, bullets, or extra text beyond the required format.

note:
1) **No type annotations or return-type hints**.
2) **Output only executable Python code**, with no tags (e.g., </...>), no markdown fences (“‘), and no explanations or comments.
3) **Wrap the solution in exactly one function**. If the function name is specified in the problem or PUBLIC TEST, use it and keep the exact parameter list. Otherwise, name the function `solution`.
4) **Match the task’s input format**. If examples indicate inputs arrive as strings, accept a string parameter and parse internally. Likewise, format outputs exactly as required (e.g., print vs return).
5) Use only Python’s standard library; do **not** rely on network access, external files, or third-party packages.
6) Ensure deterministic behavior and avoid unnecessary randomness or system calls.
7) Prefer clear, robust algorithms; handle edge cases implied by the task (empty inputs, boundary values) when reasonable.

Figure 7: Unified coding prompt suite for coding solutions. The five roles (RAW, CoT, ReAct, Reflect, Note) ensure silent planning, strict code-only output, conservative review, and precise conformance to I/O constraints.

Math Prompts and Notes

raw:
You are a competition mathematician. Solve the problem with clear step-by-step reasoning, using exact symbolic forms (prefer fractions to decimals when appropriate). End with a single final line: $\boxed{\langle answer\rangle}$. Follow **note** exactly.

cot:
First think step by step *silently* (do not print your thoughts); then present a concise, logically ordered solution. Use exact forms; avoid decimals unless requested. End with the single final line: $\boxed{\langle answer\rangle}$. Follow **note**.

react:
Analyze the problem and form a plan *silently* (do not print it); then present a full worked solution with clear steps and exact forms. If `FEEDBACK` is provided, revise the solution only when it is correct and improves adherence to **note**; otherwise keep the best prior solution. Finish with the final line: $\boxed{\langle answer\rangle}$.

reflect:
You are a competition-math **critic**. Assess the submitted solution on:
1) Mathematical correctness of the result and reasoning.
2) Output formatting (final line exactly boxed).
3) Adherence to all constraints in **note**.
If everything is correct, reply **exactly**: `ACCEPT`
Otherwise, reply **exactly**: `REVISE: <concise, actionable fixes required>`
Do not include any extra text beyond the required format.

note:
MATH Canonicalization (Top Priority)
1) Final line only: $\boxed{\langle answer\rangle}$.
2) Exact forms: reduce a/b; simplify radicals; use π; avoid decimals unless asked.
3) Numbers: no commas anywhere (9901 not 9,901; $448/15625$ not 2,240/78,125).
4) Expressions: canonical, no spaces, use ^ for exponents (x^3+3x-6). Do not prepend variables or ’=’ (prefer 5 over $x=5$).
5) MCQ: box the single capital letter only.
6) Tuples/Sets: $(a,b,\dots)$ and $\{a,b,\dots\}$ with simplified components.
7) Units: match the problem; use k° for degrees; default radians.
8) Sanity: respect domains; drop extraneous roots; include a quick plug-back check.

Figure 8: Unified math prompt suite for exact, well-formatted solutions. The five roles (RAW, CoT, ReAct, Reflect, Note) enforce silent planning, precise symbolic work, and a canonical boxed final answer. Different from coding prompts, prompts for question answering are rather similar to math prompts as they both relate to reasoning, thus we skip the demonstration here.