DebFlow: Automating Agent Creation via Agent Debate

Anonymous Authors[1]

Abstract

Large language models (LLMs) have demonstrated strong potential and impressive performance in automating the generation and optimization of workflows. However, existing approaches are marked by limited reasoning capabilities, high computational demands, and significant resource requirements. To address these issues, we propose DebFlow, a framework that employs a debate mechanism to optimize workflows and integrates reflexion to improve based on previous experiences. We evaluated our method across six benchmark datasets, including HotpotQA, MATH, and ALFWorld. Our approach achieved a 3% average performance improvement over the latest baselines, demonstrating its effectiveness in diverse problem domains. In particular, during training, our framework reduces resource consumption by 37% compared to the state-of-the-art baselines. Additionally, we performed ablation studies. Removing the Debate component resulted in a 4% performance drop across two benchmark datasets, significantly greater than the 2% drop observed when the Reflection component was removed. These findings strongly demonstrate the critical role of Debate in enhancing framework performance, while also highlighting the auxiliary contribution of reflexion to overall optimization.

1. Introduction

Large Language Models (LLMs) have demonstrated exceptional capabilities across diverse domains, such as code generation(Shinn et al., 2023), data analysis(Hong et al., 2024a), decision-making(Song et al., 2023), question answering(Zhu et al., 2024), autonomous driving(Jin et al., 2023). Historically, the development of LLMs has relied on manually crafted agents, which require significant human input for their design and orchestration. This dependency limits the scalability of LLMs, their adaptability to complex new domains, and their ability to generalize skills across diverse tasks(Tang et al., 2023). However, the history of machine learning teaches us that hand-designed solutions are eventually replaced by learned solutions.

Current research aims to develop automated frameworks for discovering efficient agentic workflows, thus minimizing human intervention. ADAS(Hu et al., 2024a) defines the entire agentic system in code. However, the efficiency limitations of the linear heuristic search algorithm of ADAS hinder its ability to generate effective workflows within a constrained number of iterations. AFlow (Zhang et al., 2024a) models the workflow as a series of interconnected LLM-invoking nodes, where each node corresponds to an LLM action, and the edges capture the logical structure, dependencies, and execution flow between these actions. AFLOW employs the Monte Carlo Tree Search (MCTS) algorithm to automatically optimize LLM agent designs. In the search process, Monte Carlo Tree Search (MCTS) often performs numerous redundant optimizations, leading to significant computational overhead. This inefficiency increases the overall cost of the search, as it spends excessive resources on exploring suboptimal or irrelevant branches in the decision tree. Consequently, the algorithm's performance can be hindered by this unnecessary expenditure of computational effort, impacting its scalability and effectiveness in large or complex problem spaces. This underscores the need for more cost-effective and efficient methods to automate the generation of agentic workflows.

Furthermore, previous works on ADAS(Hu et al., 2024a) and AFlow (Zhang et al., 2024a) primarily comprised three core components: search space, search algorithm, and evaluation. In terms of search algorithms, these approaches predominantly relied on generating workflows through a single large language model (LLM), which significantly constrains the performance to the capabilities of the individual model.

In response to these challenges, we introduce an innovative framework for automatically generating agentic workflows. We propose DebFlow, a multi-agent framework that employs a collaborative debate mechanism to optimize workflow generation and integrates reflective learning to iteratively improve performance based on previous experiences. Our

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

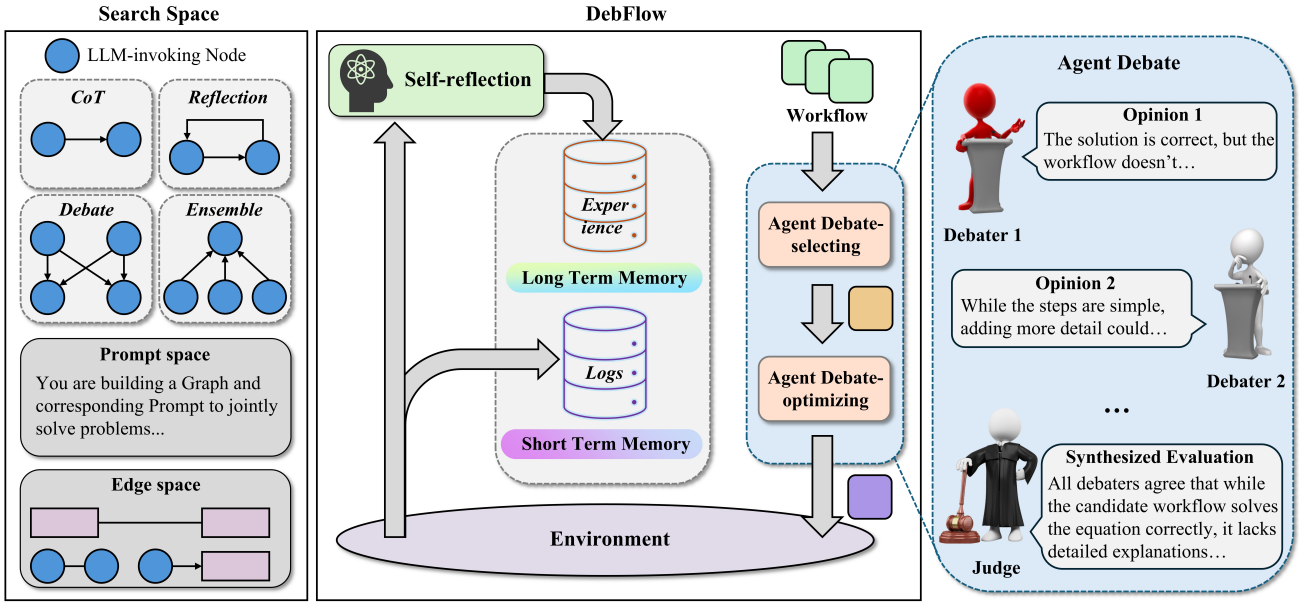

Figure 1. The overall framework of **DebFlow.** The basic unit of framework invocation is the llm-invoking node, which can be combined to form different operators. Debflow selects the most promising workflow for optimization through agent debate, then optimizes it through multi-role debate, and conducts reflection to provide direction for subsequent optimization.

work represents the first application of debate frameworks within Automated Agentic Optimization. Prior studies have predominantly utilized debate methods to augment model reasoning capabilities through direct analytical engagement with problems. For examples, LLM-Debate(Du et al., 2023), MultiPersona(Wang et al., 2023b). DebFlow uses operator nodes, that is, a set of LLM agent-invoking nodes, as the fundamental units of its search space. These operators are reusable combinations of nodes representing common agentic operations (e.g., Ensemble, Review & Revise). DebFlow optimizes LLM agents through the mechanisms of **Debate** and **reflection**. The debate-driven optimization framework facilitates comprehensive improvements in both prompts and operators. Through structured multi-agent deliberations, the system analyzes task specifications and historical performance logs to explore optimal operator configurations while simultaneously refining prompts, thereby synthesizing more efficient workflows. To avoid unnecessary branch expansions inherent in MCTS approaches, we employ **reflection** to analyze execution logs and identify failure patterns, which serve as one of the optimization factors for subsequent iterations. Furthermore, we leverage LLMs for workflow selection, incorporating both performance metrics and these derived optimization factors in the decision-making process and we employ long-term and short-term memory to maintain the structural integrity of the search process. A simplified illustration is shown in Figure1.

The key contributions of this work are as follows:

- We design the DebFlow framework that efficiently searches for novel and good-performing LLM agents via the novel mechanism of Debate, Reflection.
- Experiments across six diverse tasks show that our method discovers novel LLM agents that outperform all known human designs. Besides, DebFlow offers better cost-performance efficiency, with significant implications for real-world applications.

2. Related Work

Agentic workflow. Agentic workflows primarily involve the static execution of predefined processes, which are typically established by humans based on prior domain experience and iterative refinement. Agentic workflows can be broadly divided into two categories: general workflows and domain-specific workflows. General workflows are typically used for simple tasks, focusing on universal problem-solving approaches, such as (Wei et al., 2022a; Wang et al., 2023a; Madaan et al., 2023a). In contrast, domain-specific workflows are designed for specific fields, such as code generation (Hong et al., 2024b), data analysis (Xie et al., 2024), mathematical computation (Zhong et al., 2024), and complex question answering (Zhou et al., 2024a). Traditional agentic workflows are usually predefined manually, which limits general workflows in handling complex tasks, while domain-specific workflows are only capable of addressing tasks within their specific domains, lacking universality. As a result, new automated workflow methods have emerged,

capable of generating workflows that are both universal and effective in solving complex tasks. The agentic workflow and autonomous agents(Zhuge et al., 2024; Hong et al., 2024a; Zhang et al., 2024c; Wang et al., 2024a) represent two different paradigms of the application of LLM.

Automated Agentic Optimization. Recent advancements in automating the design of agentic workflows have explored three primary strategies: optimizing prompts, tuning hyperparameters, and refining entire workflows. Techniques for prompt optimization (Fernando et al., 2024; Yang et al., 2024) utilize large language models to enhance prompts within fixed workflows, but they often fail to generalize well to new tasks and require considerable manual effort. Hyperparameter optimization (Saad-Falcon et al., 2024) focuses on adjusting predefined parameters, yet it remains constrained by its narrow scope. On the other hand, automated workflow optimization (Li et al., 2024; Zhou et al., 2024b; Zhuge et al., 2024; Hu et al., 2024a) aims to improve the overall structure of workflows, presenting a more comprehensive approach to automation. For example, ADAS (Hu et al., 2024a) employs code-based representations and stores historical workflows in a linear structure, but its search algorithm's reliance on overly simplistic representations of past experiences significantly limits its efficiency in discovering effective workflows. AFlow (Zhang et al., 2024a) also utilizes code-based workflow representations but advances further by employing an MCTS algorithm for automated optimization, leveraging tree-structured experience and execution feedback to efficiently discover effective workflows.

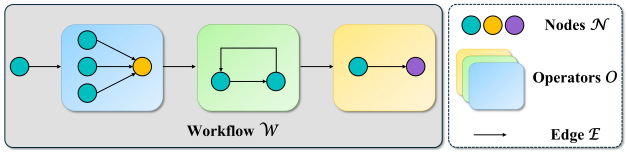

Figure 2. The visualization of notations in DebFlow

3. Problem Formulation

In this section, we provide a formal definition of DebFlow's search space and articulate the objectives of workflow optimization. For the core concept of this section, we provide an example explanation in Figure 2.

Search Space. We define the atomic units within the search space as LLM-invoking nodes $\mathcal{N}$, which can be interconnected via edges E to form more widely recognized operators $\mathcal{O}$, eventually being integrated to constitute comprehensive workflows $\mathcal{W}$. Each node is represented as follows:

$$N_i = (M_i, P_i, \tau_i),\ P_i \in \mathcal{P},\ \tau_i \in [0, 1], \tag{1}$$

where M_i represents an LLM instance, P_i represents the associated prompt, with P denoting the feasible prompt space and τ_i is the temperature parameter. The operators are composed of nodes and edges, represented as follows:

$$O_j = (\mathcal{N}_j^o, \mathcal{E}_j^o), \mathcal{N}_j^o = \{N_1, \ldots, N_n\}, \mathcal{E}_j^o \subseteq E, \tag{2}$$

where $\mathcal{N}_j^o$ is a subset of the invoking nodes, and $\mathcal{E}_j^o$ represents the connection between the nodes, which governs the sequence of execution and E represents the collection of connectivity patterns established between nodes and nodes, nodes and operators, as well as between operators and operators. The overall agentic workflow $\mathcal{W}$ is defined as:

$$\mathcal{W} = (O^S, \mathcal{E}^a) = (\mathcal{N}^S, \mathcal{E}), O^S = \{O_1, \ldots, O_m\}, \mathcal{E}^a, \mathcal{E} \subseteq E, \tag{3}$$

where $\mathcal{O}^S \subseteq \mathcal{O}$, $\mathcal{N}^S \subseteq \mathcal{N}$, m represents the number of operators in $\mathcal{W}$.

Automated Workflow Optimization. Given a task domain T and an performance evaluator function U, the goal of workflow optimization is to discover a workflow $\mathcal{W}$ that maximizes $U(\mathcal{W}, T)$, the objective function is defined as:

$$\mathcal{W}^* = \arg\max_{\mathcal{W} \in S} U(\mathcal{W}, T) = \arg\max_{\mathcal{N}^S \subseteq \mathcal{N}, \mathcal{E} \subseteq E} U\left((\mathcal{N}^S, \mathcal{E}), T\right), \tag{4}$$

where $\mathcal{N}$ represents the feasible space of invoking nodes.

4. DebFlow Framework: Automated Agent Generation

In this section, we describe the framework for automating the generation workflows using the **DebFlow** system. As shown in Figure 1, we utilize agent debate and reflexion mechanisms to facilitate automated exploration of optimal workflow configurations.

4.1. Agent debate

Figure 1 illustrates the general framework of Agent Debate, where debaters and a judge are involved in a debate to resolve problems. Generally, the Agent Debate framework is composed of two roles, which are elaborated as follows:

Debater. In the framework, there are N debaters, denoted $D = \{D_i\}_{i=1}^N$. During each iteration of the debate, the debaters D_i present their arguments sequentially in a predetermined order. The argument of each debater is formulated based on the accumulated history of the debate H, such that $D_i(H) = h$. The Debaters utilizes a structured dialectical process featuring proponents and opponents. When one side proposes a solution, the opposing side critically evaluates and thinks about it. Moreover, the opposing side integrates the insights from the proposed solution to refine and enhance its own approach. This process allows each side to absorb the strengths of the other's solution while addressing its weaknesses, ultimately leading to a more robust and improved solution. This iterative exchange fosters a dynamic

and collaborative environment, where each debater's contribution not only challenges, but also enriches the overall discourse.

Judge. we introduce a judge J to supervise and regulate the entire debate process. After each round of debate, the judge J reviews the proposals of both the proponents and opponents, summarizing their respective strengths and weaknesses. The judge J then evaluates which side currently has the advantage and determines whether the proposed solution can be considered the optimal workflow in this round. If the solution is deemed optimal, the process skips subsequent rounds and proceeds to the next phase. If not, another round of debate is initiated. If the maximum number of debate rounds is reached without identifying an optimal workflow, the judge J selects the best solution from the accumulated proposals based on the historical outcomes of the debate. This structured approach ensures a balanced and efficient decision-making process, guided by continuous evaluation and refinement.

4.2. Selecting candidate workflows

The Selection component of our framework is designed to choose the most appropriate workflow for a given task. In analogous fashion, we implement agent debate to realize this objective. In debate competitions, the selection phase can be analogized to choosing the most powerful arguments or strategies available. Debaters must select from multiple potential arguments those most likely to persuade judges or audiences. This process parallels the selection of the most promising nodes for further exploration in Monte Carlo Tree Search (MCTS). It draws on long-memory, which captures historical insights from past workflow performances, to avoid repeating former errors. Additionally, it considers short memory, focusing on individual workflow failures to exclude those with a history of underperformance. The component also ensures that its selections align with the specific requirements of the current task, thereby guaranteeing that the chosen workflow is well-suited to achieve the task's objectives.

4.3. Reflexion

In our framework, the Large Language Model (LLM) serves as a critical reflection model, generating detailed verbal feedback to guide future iterations. This feedback is instrumental in refining the workflow and enhancing its performance. Post-debate, when the optimal workflow is identified, it is executed on the dataset, and the instances of failure are meticulously recorded. The reflection model then analyzes these failed cases and the workflow itself to determine the root causes of the failures, providing valuable insights for subsequent optimization efforts.

For instance, if a workflow execution results in unsuccessful data points, the reflection model dissectes the workflow to pinpoint the specific steps that contributed to these outcomes. This nuanced feedback is subsequently integrated into the current workflow's nodes, thereby informing and improving future decision-making processes. Through this iterative mechanism, our framework systematically enhances the robustness and efficiency of the workflows, ensuring continuous improvement and adaptation to diverse challenges.

5. Experiments

5.1. Experiment Setup

Tasks and Benchmarks. We conduct experiments on six representative tasks covering four domains:(1)reading comprehension, HotpotQA(Yang et al., 2018), DROP(Dua et al., 2019); (2)math reasoning, MATH(Hendrycks et al., 2021); (3)code generation, HumanEval(Chen et al., 2021) and MBPP(Austin et al., 2021); (4)embodied, ALFWorld(Shridhar et al., 2020). Following prior studies such as (Hu et al., 2024a) and (Shinn et al., 2023), we extracted 1,000 random samples each from the HotpotQA and DROP datasets. We also examined 617 problems from the MATH dataset, specifically choosing difficulty level 5 questions across four categories: Combinatorics & Probability, Number Theory, Pre-algebra, and Pre-calculus, consistent with the approach taken by (Hong et al., 2024a).

Baselines. We compare DebFlow with two series of agentic baselines:(1) manually designed workflows, IO (direct LLM invocation), Chain-of-Thought(Wei et al., 2022b), Self-Consistency (SC)(Wang et al., 2022b), MultiPersona(Wang et al., 2024b); (2) autonomous workflows, ADAS(Hu et al., 2024a), AFlow(Zhang et al., 2024a).

LLM Backbones. In our experimental framework, DebFlow utilizes different models for optimization and execution. We employ GPT-4o-mini as the optimizer and use models: GPT-4o-mini-0718, Claude-3.5-sonnet-0620, GPT-4o-0513 as executors. All models are accessed via APIs. We set the temperature to 0 for all models. We set iteration rounds to 20 for AFLOW and 10 for DebFlow. For ADAS, we use Claude-3.5-sonnet as the optimizer and GPT-4o-mini as the executor, with the iteration rounds set to 30.

Evaluation Metrics. For quantitative assessment of model performance, we employ task-specific evaluation criteria across our experimental datasets. In mathematical reasoning tasks (GSM8K and MATHlv5*), solution accuracy is measured via the Solve Rate percentage metric. For programming proficiency evaluation (HumanEval and MBPP), we utilize the pass@1 metric, following the methodology established by (Chen et al., 2021). Question-answering performance (HotpotQA and DROP) is assessed through F1 Score computation. To comprehensively evaluate methodological efficiency, we conduct token consumption analysis

Method	MATH	HotpotQA	HumanEval	MBPP	ALFWorld	DROP	Avg.
IO (GPT-4o-mini)	47.8	68.1	87.0	71.8	38.7	68.3	63.6
CoT (Wei et al., 2022c)	48.8	67.9	88.6	71.8	39.9	78.5	65.9
CoT SC (Wang et al., 2022a)	47.9	68.9	88.6	73.6	40.5	78.8	66.4
MultiPersona (Wang et al., 2023c)	50.8	69.2	88.3	73.1	39.1	74.4	68.8
Self Refine (Madaan et al., 2023b)	46.1	60.8	87.8	69.8	40.0	70.2	62.5
ADAS (Hu et al., 2024b)	43.1	64.5	82.4	53.4	47.7	76.6	61.3
AFlow (Zhang et al., 2024b)	53.8	73.5	90.9	81.4	59.2	80.3	73.2
DebFlow(Ours)	**55.5**	**75.4**	**91.5**	**82.4**	**62.3**	**80.7**	**74.6**

Table 1. Comparison of performance between manually designed methods and workflow generated by automated workflow optimization methods. All methods are executed with GPT-4o-mini on divided test set, and we tested it three times and reported it on the average.

Method	MATH	HotpotQA	HumanEval	MBPP	DROP	Avg.
$\text{AFlow}_{\text{gpt-4o-mini}}$ (Zhang et al., 2024b)	4.76	5.12	0.84	5.56	3.36	3.93
$\text{DebFlow}_{\text{gpt-4o-mini}}$	**4.23**	**3.34**	**0.61**	**1.78**	**1.24**	**2.24**
$\text{AFlow}_{\text{deepseek}}$ (Zhang et al., 2024b)	2.76	3.42	0.54	0.88	2.02	1.93
$\text{DebFlow}_{\text{deepseek}}$	**2.10**	**2.56**	**0.34**	**0.62**	**1.21**	**1.43**

Table 2. Training API costs. All the baselines employ GPT-4o-mini as the optimizer and the executor.

across all datasets, constructing Pareto-optimal frontiers to elucidate the performance-cost equilibrium among diverse approaches.

5.2. EXPERIMENTAL RESULTS AND ANALYSIS

Main Results. Table 1 demonstrates that DebFlow outperforms existing hand-crafted or automated agentic workflows across six benchmarks. Specifically, On the embodied benchmark ALFWorld, DebFlow achieves the optimal 62.3%, outperforming the secondbest AFLOW by 3.1%. On the MATH benchmark, it exceeds IO(gpt-4o-mini) by 7.7% and surpasses the SOTA baseline AFlow by 1.7%. Across six datasets in QA, Code, Embodied and Math domains, Debflow surpasses all manually crafted workflows and demonstrates marginal improvements compared to automatically generated workflows.

Cost Analysis. We demonstrate the resource-friendly nature of DebFlow's agentic automation system in training API costs. During the search process, we conducted three trials and averaged the results across various benchmarks, employing GPT-4o-mini as the optimizer and the executor. As shown in Table 2, Across various benchmarks, Debflow consistently demonstrates lower training costs compared to Aflow. Notably, in the MBPP benchmark, DebFlow incurred a cost of 1.78$, while AFlow required 5.56$, representing a 68% reduction in expenditure. Overall, Debflow demonstrates an average reduction in consumption of 43% compared to AFlow.

Debate Study. Next, we analyze the impact of the number of debating agents in the debate. In Figure 3, we increase the number of debating agents, while maintaining a fixed debate length of two rounds. It seems intuitive that increasing the number of debaters would enhance diversity of thought and subsequently improve performance. However, our experiments show an increase in the number of debaters has resulted in varying degrees of performance reduction. As the number of debating agents increases, the mathematical performance demonstrates a non-monotonic trend, initially improving before subsequently declining. This phenomenon can be attributed to the fact that an expansion in the number of participants corresponds to increased textual length and complexity. Language model-based debaters exhibit a propensity to lose track of perspectives articulated by other agents during extended multi-party discussions, compromising their ability to effectively incorporate all relevant viewpoints.

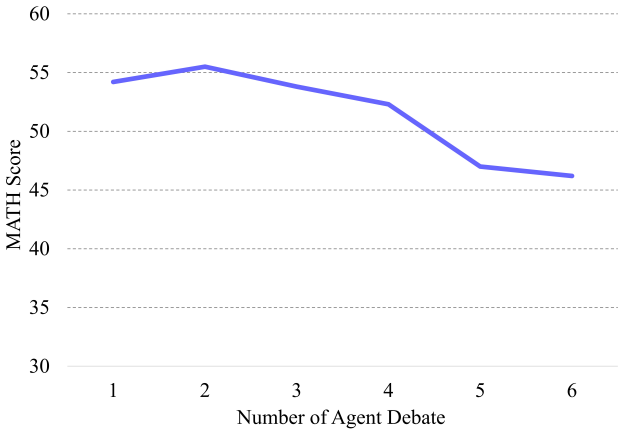

Figure 3. Effect of Number of agent debate

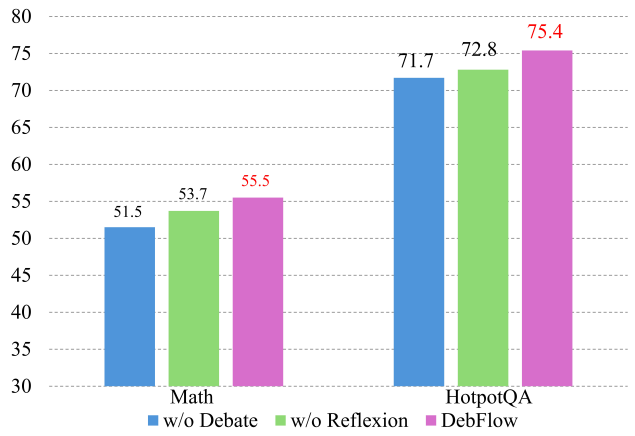

Figure 4. The ablation study of DebFlow

Ablation Study. To evaluate the contribution of each component in our proposed method, we conducted a series of ablation studies. We perform an ablation study on two variants of DebFlow: w/o Debate, where agent debate is removed and replaced with an LLM as an optimizer to create new workflows while randomly selecting candidate workflows; w/o reflexion, where self-reflection is removed. Figure 4 presents the results of our experiments. The complete framework achieved strong performance across both datasets, obtaining 55.5% accuracy on MATH and 75.4% on HotpotQA. When the debate component was removed (w/o debate), performance decreased notably to 51.4% on MATH and 71.5% on HotpotQA, demonstrating the critical role of multi-agent deliberation in enhancing reasoning capabilities. Similarly, ablating the reflection mechanism (w/o reflection) resulted in performance drops to 53.7% on MATH and 72.8% on HotpotQA. These results confirm that both components contribute substantially to the overall effectiveness of our approach, with the debate component showing a slightly larger impact on performance across both datasets.

Case Study. As shown in Figure 5, beginning with a single node (Node 1, score 0.4789), each iteration involved precisely one targeted modification or addition. First, in Node 2 (score 0.4846), an additional review step was introduced to verify solutions before returning results, slightly improving accuracy. Next, Node 3 (score 0.4854) incorporated a ”Programmer” operator that automatically writes and executes Python code to solve problems, further optimizing the resolution process. Subsequently, Node 4 (score 0.4922) added a self-ensemble step to generate multiple solutions and select the best one, ensuring its robustness and accuracy. In parallel, certain exploratory branches (such as Nodes 5 and 7) attempted direct modifications to already generated complex solutions, but failed to improve accuracy due to insufficient further reasoning. Finally, Node 8 (score 0.5546)

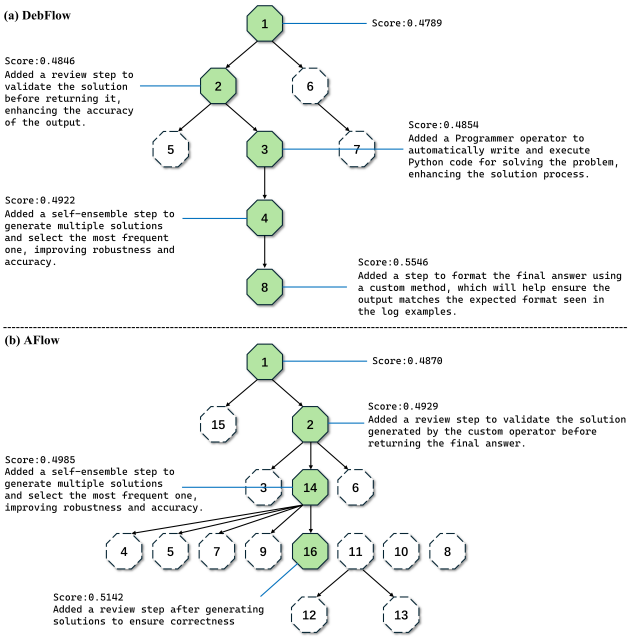

Figure 5. Tree-structured iteration process of DebFlow and AFlow on MATH

implemented custom formatting operations, making the output more consistent with expected formats and achieving the highest accuracy to date. Similarly, Figure 5 demonstrates the tree structure iteration process of aflow on math. Node 3 introduces the ”review” operation, which is already present in Node 2. This redundancy leads to suboptimal performance. Similarly, Node 6 adds the ”self-consistency” effect, aligning with Node 14. However, instead of improving performance, this change results in a performance decline, forcing AFlow to re-optimize Node 2 to achieve the outcome of Node 14. This example demonstrates that AFlow makes numerous erroneous attempts during the optimization process. These errors arise from the incorrect selection of the node to optimize and misguided judgments about the optimization direction, leading to an increase in cost. In contrast, our DebFlow can achieve precise optimization. Detailed comparison of the workflow structures can be found in Appendix B

6. CONCLUSION

In conclusion, we introduced DebFlow, a novel framework that optimizes workflows using agent debate and reflection mechanisms. This approach offers a significant improvement in both performance and efficiency over existing methods. Future work will explore extending DebFlow to handle more complex, multi-domain tasks and further reduce its computational cost. By utilizing LLM agent call nodes as basic building blocks and driven by structured multi-agent debates, DebFlow efficiently searches and optimizes

workflows based on task specifications and historical execution feedback. Experimental results demonstrate that DebFlow achieves an average performance improvement of approximately 3% across six different datasets, outperforming existing manual design and automated methods in mathematical reasoning, question answering, code generation, and entity tasks. Meanwhile, cost analysis reveals that DebFlow reduces resource consumption by 37% during the training process compared to state-of-the-art baselines, further validating its cost-effectiveness in practical applications. Ablation studies also confirm the critical role of the debate mechanism in enhancing overall performance, with the removal of the debate module resulting in more significant performance degradation compared to relying solely on reflection. Overall, DebFlow provides a more efficient, energy-saving, and adaptive solution for automatic agent generation, with future work potentially exploring more complex reasoning strategies and extensions to cross-domain applications.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C. J., Terry, M., Le, Q. V., and Sutton, C. Program synthesis with large language models. *ArXiv*, abs/2108.07732, 2021. URL https://api.semanticscholar.org/CorpusID:237142385.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pondé, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D. W., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Babuschkin, I., Balaji, S., Jain, S., Carr, A., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M. M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374, 2021. URL https://api.semanticscholar.org/CorpusID:235755472.

Du, Y., Li, S., Torralba, A., Tenenbaum, J. B., and Mordatch, I. Improving factuality and reasoning in language models through multiagent debate. *ArXiv*, abs/2305.14325, 2023. URL https://api.semanticscholar.org/CorpusID:258841118.

Dua, D., Wang, Y., Dasigi, P., Stanovsky, G., Singh, S., and Gardner, M. DROP: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In Burstein, J., Doran, C., and Solorio, T. (eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 2368–2378, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1246. URL https://aclanthology.org/N19-1246/.

Fernando, C., Banarse, D. S., Michalewski, H., Osindero, S., and Rocktäschel, T. Promptbreeder: Self-referential self-improvement via prompt evolution. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=9ZxnPZGmPU.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D. X., and Steinhardt, J. Measuring mathematical problem solving with the math dataset. *ArXiv*, abs/2103.03874, 2021. URL https://api.semanticscholar.org/CorpusID:232134851.

Hong, S., Lin, Y., Liu, B., Wu, B., Li, D., Chen, J., Zhang, J., Wang, J., Zhang, L., Zhuge, M., Guo, T., Zhou, T., Tao, W., Wang, W., Tang, X., Lu, X., Liang, X., Fei, Y., Cheng, Y., Gou, Z., Xu, Z., Wu, C., Zhang, L., Yang, M., and Zheng, X. Data interpreter: An llm agent for data science. *arXiv preprint*, 2024a.

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Wang, J., Zhang, C., Wang, Z., Yau, S. K. S., Lin, Z., Zhou, L., Ran, C., Xiao, L., Wu, C., and Schmidhuber, J. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024b.

Hu, S., Lu, C., and Clune, J. Automated design of agentic systems. *ArXiv*, abs/2408.08435, 2024a. URL https://api.semanticscholar.org/CorpusID:271892234.

Hu, S., Lu, C., and Clune, J. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024b.

Jin, Y., Yang, R., Yi, Z., Shen, X., Peng, H., Liu, X., Qin, J., Li, J., Xie, J., Gao, P., Zhou, G., and Gong, J. Surrealdriver: Designing llm-powered generative driver agent framework based on human drivers' driving-thinking data. *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 966–971,

2023. URL https://api.semanticscholar.org/CorpusID:271329438.

Li, Z., Xu, S., Mei, K., Hua, W., Rama, B., Raheja, O., Wang, H., Zhu, H., and Zhang, Y. Autoflow: Automated workflow generation for large language model agents. *ArXiv*, abs/2407.12821, 2024. URL https://api.semanticscholar.org/CorpusID:271270428.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A., and Clark, P. Self-refine: Iterative refinement with self-feedback. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594. Curran Associates, Inc., 2023a.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Welleck, S., Majumder, B. P., Gupta, S., Yazdanbakhsh, A., and Clark, P. Self-refine: Iterative refinement with self-feedback. *ArXiv*, abs/2303.17651, 2023b. URL https://api.semanticscholar.org/CorpusID:257900871.

Saad-Falcon, J., Lafuente, A. G., Natarajan, S., Maru, N., Todorov, H., Guha, E. K., Buchanan, E. K., Chen, M., Guha, N., Ré, C., and Mirhoseini, A. Archon: An architecture search framework for inference-time techniques. *ArXiv*, abs/2409.15254, 2024. URL https://api.semanticscholar.org/CorpusID:272827424.

Shinn, N., Cassano, F., Labash, B., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: language agents with verbal reinforcement learning. In *Neural Information Processing Systems*, 2023. URL https://api.semanticscholar.org/CorpusID:258833055.

Shridhar, M., Yuan, X., Côté, M.-A., Bisk, Y., Trischler, A., and Hausknecht, M. J. Alfworld: Aligning text and embodied environments for interactive learning. *ArXiv*, abs/2010.03768, 2020. URL https://api.semanticscholar.org/CorpusID:222208810.

Song, C. H., Sadler, B. M., Wu, J., Chao, W.-L., Washington, C., and Su, Y. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 2986–2997, 2023. doi: 10.1109/ICCV51070.2023.00280.

Tang, N., Yang, C., Fan, J., and Cao, L. Verifai: Verified generative ai. *ArXiv*, abs/2307.02796, 2023. URL https://api.semanticscholar.org/CorpusID:259360404.

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024a. ISSN 2835-8856.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022a.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E. H., and Zhou, D. Self-consistency improves chain of thought reasoning in language models. *ArXiv*, abs/2203.11171, 2022b. URL https://api.semanticscholar.org/CorpusID:247595263.

Wang, X., Wei, J., Schuurmans, D., Le, Q. V., Chi, E. H., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023a. URL https://openreview.net/forum?id=1PL1NIMMrw.

Wang, Z., Mao, S., Wu, W., Ge, T., Wei, F., and Ji, H. Unleashing the emergent cognitive synergy in large language models: A task-solving agent through multi-persona self-collaboration. In *North American Chapter of the Association for Computational Linguistics*, 2023b. URL https://api.semanticscholar.org/CorpusID:259765919.

Wang, Z., Mao, S., Wu, W., Ge, T., Wei, F., and Ji, H. Unleashing the emergent cognitive synergy in large language models: A task-solving agent through multi-persona self-collaboration. *arXiv preprint arXiv:2307.05300*, 2023c.

Wang, Z., Mao, S., Wu, W., Ge, T., Wei, F., and Ji, H. Unleashing the emergent cognitive synergy in large language models: A task-solving agent through multi-persona self-collaboration. In Duh, K., Gomez, H., and Bethard, S. (eds.), *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 257–279, Mexico City, Mexico, June 2024b. Association for Computational Linguistics. doi: 10.18653/v1/2024.naacl-long.15. URL https://aclanthology.org/2024.naacl-long.15/.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., ichter, b., Xia, F., Chi, E., Le, Q. V., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language

models. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A. (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837. Curran Associates, Inc., 2022a. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E. H., Le, Q. V., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022b. Curran Associates Inc. ISBN 9781713871088.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022c.

Xie, Y., Luo, Y., Li, G., and Tang, N. Haichart: Human and ai paired visualization system. *ArXiv*, abs/2406.11033, 2024.

Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q. V., Zhou, D., and Chen, X. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Bb4VGOWELI.

Yang, Z., Qi, P., Zhang, S., Bengio, Y., Cohen, W., Salakhutdinov, R., and Manning, C. D. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In Riloff, E., Chiang, D., Hockenmaier, J., and Tsujii, J. (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

Zhang, J., Xiang, J., Yu, Z., Teng, F., Chen, X., Chen, J., Zhuge, M., Cheng, X., Hong, S., Wang, J., Zheng, B., Liu, B., Luo, Y., and Wu, C. Aflow: Automating agentic workflow generation. *ArXiv*, abs/2410.10762, 2024a. URL https://api.semanticscholar.org/CorpusID:273345847.

Zhang, J., Xiang, J., Yu, Z., Teng, F., Chen, X., Chen, J., Zhuge, M., Cheng, X., Hong, S., Wang, J., et al. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*, 2024b.

Zhang, J., Zhao, C., Zhao, Y., Yu, Z., He, M., and Fan, J. Mobileexperts: A dynamic tool-enabled agent team in mobile devices. *ArXiv*, abs/2407.03913, 2024c.

Zhong, Q., Wang, K., Xu, Z., Liu, J., Ding, L., Du, B., and Tao, D. Achieving ¿97 *arXiv preprint arXiv:2404.14963*, 2024.

Zhou, A., Yan, K., Shlapentokh-Rothman, M., Wang, H., and Wang, Y.-X. Language agent tree search unifies reasoning, acting, and planning in language models. In *Forty-first International Conference on Machine Learning*, 2024a. URL https://openreview.net/forum?id=njwv9BsGHF.

Zhou, W., Ou, Y., Ding, S., Li, L., Wu, J., Wang, T., Chen, J., Wang, S., Xu, X., Zhang, N., Chen, H., and Jiang, Y. E. Symbolic learning enables self-evolving agents. *ArXiv*, abs/2406.18532, 2024b. URL https://api.semanticscholar.org/CorpusID:270737580.

Zhu, J.-P., Cai, P., Xu, K., Li, L., Sun, Y., Zhou, S., Su, H., Tang, L., and Liu, Q. Autotqa: Towards autonomous tabular question answering through multi-agent large language models. *Proc. VLDB Endow.*, 17(12): 3920–3933, August 2024. ISSN 2150-8097. doi: 10.14778/3685800.3685816. URL https://doi.org/10.14778/3685800.3685816.

Zhuge, M., Wang, W., Kirsch, L., Faccio, F., Khizbullin, D., and Schmidhuber, J. GPTSwarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*, 2024.

A. Appendix

A.1. BASIC NODE

```
class ActionNode:
    async def fill(
        self,
        context, #:param context: Everything we should know when filling node.
        llm,     #:param llm: Large Language Model with pre-defined system message.
        ...)
        return self
```

A.2. INITIAL WORKFLOW STRUCTURE

```
class Workflow:
    def __init__(
        self,
        name: str,
        llm_config,
        dataset: DatasetType,
    ) -> None:
        self.name = name
        self.dataset = dataset
        self.llm = create_llm_instance(llm_config)
        self.llm.cost_manager = CostManager()
        self.custom = operator.Custom(self.llm)

    async def __call__(self, problem: str):
        """
        Implementation of the workflow
        """
        solution = await self.custom(input=problem, instruction="")
        return solution['response'], self.llm.cost_manager.total_cost
```

A.3. WORKFLOW OPTIMIZE PROMPT

```
workflow_optimize_prompt = """
First, provide optimization ideas. Only one detail point can be modified at a time, and
    no more than 5 lines of code may be changed per modification--extensive modifications
    are strictly prohibited to maintain project focus!
When introducing new functionalities in the graph, please make sure to import the
    necessary libraries or modules yourself, except for operator, prompt_custom,
    create_llm_instance, and CostManage, which have already been automatically imported.
**Under no circumstances should Graph output None for any field.**
Use custom methods to restrict your output format, rather than using code (outside of the
    code, the system will extract answers based on certain rules and score them).
It is very important to format the Graph output answers, you can refer to the standard
    answer format in the log.

Here's an example of using the `custom` method in graph:
```
You can write your own prompt in <prompt>prompt_custom</prompt> and then use it in the
    Custom method in the graph
response = await self.custom(input=problem, instruction=prompt_custom.XXX_PROMPT)
# You can also concatenate previously generated string results in the input to provide
    more comprehensive contextual information.
# response = await self.custom(input=problem+f"xxx:{xxx}, xxx:{xxx}",
    instruction=prompt_custom.XXX_PROMPT)
# The output from the Custom method can be placed anywhere you need it, as shown in the
    example below
solution = await self.generate(problem=f"question:{problem}, xxx:{response['response']}")
```
```

```
Note: In custom, the input and instruction are directly concatenated(instruction+input),
    and placeholders are not supported. Please ensure to add comments and handle the
    concatenation externally.\n

**Introducing multiple operators at appropriate points can enhance performance. If you
    find that some provided operators are not yet used in the graph, try incorporating
    them. Be careful not to import operators that are not included in the operator,
    otherwise the program will fail.**

please reconstruct and optimize them. You can add, modify, or delete nodes, parameters,
    or prompts. Include your single modification in XML tags in your reply. Ensure they
    are complete and correct to avoid runtime failures.
When optimizing, you can incorporate critical thinking methods like review, revise,
    ensemble (generating multiple answers through different/similar prompts, then
    voting/integrating/checking the majority to obtain a final answer), selfAsk, etc.
    Consider
Python's loops (for, while, list comprehensions), conditional statements (if-elif-else,
    ternary operators),
or machine learning techniques (e.g., linear regression, decision trees, neural networks,
    clustering). The graph
complexity should not exceed 10. Use logical and control flow (IF-ELSE, loops) for a more
    enhanced graphical
representation.Ensure that all the prompts required by the current graph from
    prompt_custom are included.Exclude any other prompts.
Output the modified graph and all the necessary Prompts in prompt_custom (if needed).
The prompt you need to generate is only the one used in `prompt_custom.XXX` within
    Custom. Other methods already have built-in prompts and are prohibited from being
    generated. Only generate those needed for use in `prompt_custom`; please remove any
    unused prompts in prompt_custom.
the generated prompt must not contain any placeholders.
Considering information loss, complex graphs may yield better results, but insufficient
    information transmission can omit the solution. It's crucial to include necessary
    context during the process.
"""
```

A.4. AGENT DEBATE

```
debate_prompt_meta_1 = """You are a debater. Hello and welcome to the debate. It's not
    necessary to fully agree with each other's perspectives, as our objective is to find
    the correct answer.
The debate topic is how to optimize the Graph and corresponding Prompt. You should
    analyze log data and come up with an optimization plan.

Below are the logs of some results with the aforementioned Graph that performed well but
    encountered errors, which can be used as references for optimization:
{log}

It is very important to format the Graph output answers, you can refer to the standard
    answer format in the log.
"""

debate_prompt_meta_2 ="""
Below is my answer based on the initial graph and prompt. Do you agree with my
    perspective? You have to consider whether my answer can solve the problems in the
    logs.
You must make further optimizations and improvements based on this graph. The modified
    graph must differ from the provided example, and the specific differences should be
    noted within the <modification>xxx</modification> section.
<sample>
    <modification>{modification}</modification>
    <graph>{graph}</graph>
    <prompt>{prompt}</prompt>(only prompt_custom)
</sample>
"""
```

```
Debate_prompt = debate_prompt_meta_1 + debate_prompt_meta_2

moderator_prompt_meta_1 = """
You are a moderator. There will be two debaters involved in a debate.
They will present their answers and discuss their perspectives on the following topic:
The debate topic is how to optimize the Graph and corresponding Prompt.
<initial>
    <graph>{graph}</graph>
    <prompt>{prompt}</prompt>
</initial>

At the end of each round, you will evaluate answers and decide which is correct.
"""

moderator_prompt_meta_2 = """
Now the round of debate for both sides has ended.
You have to consider which side of the workflow will not have problems in the logs after
    execution.

Affirmative side arguing:
<aff_ans>
    <modification>{aff_modification}</modification>
    <graph>{aff_graph}</graph>
    <prompt>{aff_prompt}</prompt>
</aff_ans>

Negative side arguing:
<neg_ans>
    <modification>{neg_modification}</modification>
    <graph>{neg_graph}</graph>
    <prompt>{neg_prompt}</prompt>
</neg_ans>

You, as the moderator, will evaluate both sides' answers and determine if there is a
    clear preference for an answer candidate. If so, please output your supporting
    'affirmative' or 'negative' side and give the final answer that you think is correct,
    and the debate will conclude. If not, just output 'No', the debate will continue to
    the next round.
for examples: 'affirmative' , 'negative', 'No'
"""

moderator_prompt = moderator_prompt_meta_1 + moderator_prompt_meta_2

judge = """
Now the round of debate for both sides has ended.
You have to consider which side of the workflow will not have problems in the logs after
    execution.
Affirmative side arguing:
<aff_ans>
    <modification>{aff_modification}</modification>
    <graph>{aff_graph}</graph>
    <prompt>{aff_prompt}</prompt>
</aff_ans>

Negative side arguing:
<neg_ans>
    <modification>{neg_modification}</modification>
    <graph>{neg_graph}</graph>
    <prompt>{neg_prompt}</prompt>
</neg_ans>

As a judge, the current round has ended. You must choose one of the affirmative and
    negative as your final choice. Please base your judgment on the original graph and
    the revisions of both affirmative and negative.
```

```
If you choose affirmative, please output 'affirmative'. If you choose negative, please
    output 'negative'.
for examples: 'affirmative' , 'negative'

Please strictly output format, do not output irrelevant content.
"""
```

A.5. OPERATORS

```
class Programmer(Operator):
    async def exec_code(self, code, timeout=30):
        loop = asyncio.get_running_loop()
        with concurrent.futures.ProcessPoolExecutor(max_workers=1) as executor:
            try:
                # Submit run_code task to the process pool
                future = loop.run_in_executor(executor, run_code, code)
                # Wait for the task to complete or timeout
                result = await asyncio.wait_for(future, timeout=timeout)
                return result
            except asyncio.TimeoutError:
                # Timeout, attempt to shut down the process pool
                executor.shutdown(wait=False, cancel_futures=True)
                return "Error", "Code execution timed out"
            except Exception as e:
                return "Error", f"Unknown error: {str(e)}"

    async def code_generate(self, problem, analysis, feedback, mode):
        prompt = PYTHON_CODE_VERIFIER_PROMPT.format(
            problem=problem,
            analysis=analysis,
            feedback=feedback
        )
        response = await self._fill_node(CodeGenerateOp, prompt, mode,
            function_name="solve")
        return response

    @retry(stop=stop_after_attempt(3), wait=wait_fixed(2))
    async def __call__(self, problem: str, analysis: str = "None"):
        code = None
        output = None
        feedback = ""
        for i in range(3):
            code_response = await self.code_generate(problem, analysis, feedback,
                mode="code_fill")
            code = code_response.get("code")
            if not code:
                return {"code": code, "output": "No code generated"}
            status, output = await self.exec_code(code)
            if status == "Success":
                return {"code": code, "output": output}
            else:
                print(f"Execution error on attempt {i + 1}, error message: {output}")
                feedback = (
                    f"\nThe result of the error from the code you wrote in the previous
                        round:\n"
                    f"Code: {code}\n\nStatus: {status}, {output}"
                )
        return {"code": code, "output": output}


class ScEnsemble(Operator):
    async def __call__(self, solutions: List[str], problem: str):
        answer_mapping = {}
        solution_text = ""
```

```
        for index, solution in enumerate(solutions):
            answer_mapping[chr(65 + index)] = index
            solution_text += f"{chr(65 + index)}: \n{str(solution)}\n\n\n"

        prompt = SC_ENSEMBLE_PROMPT.format(problem=problem, solutions=solution_text)
        response = await self._fill_node(ScEnsembleOp, prompt, mode="xml_fill")

        answer = response.get("solution_letter", "")
        answer = answer.strip().upper()

        return {"response": solutions[answer_mapping[answer]]}

class CustomCodeGenerate(Operator):
    async def __call__(self, problem, entry_point, instruction):
        prompt = instruction + problem
        response = await self._fill_node(GenerateOp, prompt, mode="code_fill",
            function_name=entry_point)
        return response

class Test(Operator):
    def exec_code(self, solution, entry_point):

        test_cases = extract_test_cases_from_jsonl(entry_point, dataset="MBPP")

        fail_cases = []
        for test_case in test_cases:
            test_code = test_case_2_test_function(solution, test_case, entry_point)
            try:
                exec(test_code, globals())
            except AssertionError as e:
                exc_type, exc_value, exc_traceback = sys.exc_info()
                tb_str = traceback.format_exception(exc_type, exc_value, exc_traceback)
                with open("tester.txt", "a") as f:
                    f.write("test_error of " + entry_point + "\n")
                error_infomation = {
                    "test_fail_case": {
                        "test_case": test_case,
                        "error_type": "AssertionError",
                        "error_message": str(e),
                        "traceback": tb_str,
                    }
                }
                fail_cases.append(error_infomation)
            except Exception as e:
                with open("tester.txt", "a") as f:
                    f.write(entry_point + " " + str(e) + "\n")
                return {"exec_fail_case": str(e)}
        if fail_cases != []:
            return fail_cases
        else:
            return "no error"

    async def __call__(
        self, problem, solution, entry_point, test_loop: int = 3
    ):
        for _ in range(test_loop):
            result = self.exec_code(solution, entry_point)
            if result == "no error":
                return {"result": True, "solution": solution}
            elif "exec_fail_case" in result:
                result = result["exec_fail_case"]
                prompt = REFLECTION_ON_PUBLIC_TEST_PROMPT.format(
                    problem=problem,
                    solution=solution,
                    exec_pass=f"executed unsuccessfully, error: \n {result}",
```

```
                    test_fail="executed unsucessfully",
                )
                response = await self._fill_node(ReflectionTestOp, prompt,
                    mode="code_fill")
                solution = response["reflection_and_solution"]
            else:
                prompt = REFLECTION_ON_PUBLIC_TEST_PROMPT.format(
                    problem=problem,
                    solution=solution,
                    exec_pass="executed successfully",
                    test_fail=result,
                )
                response = await self._fill_node(ReflectionTestOp, prompt,
                    mode="code_fill")
                solution = response["reflection_and_solution"]

        result = self.exec_code(solution, entry_point)
        if result == "no error":
            return {"result": True, "solution": solution}
        else:
            return {"result": False, "solution": solution}

class AnswerGenerate(Operator):
    async def __call__(self, input: str, mode: str = None) -> Tuple[str, str]:
        prompt = ANSWER_GENERATION_PROMPT.format(input=input)
        response = await self._fill_node(AnswerGenerateOp, prompt, mode="xml_fill")
        return response

class Review(Operator):
    async def __call__(self, problem, solution, mode: str = None):
        prompt = REVIEW_PROMPT.format(problem_description=problem, solution=solution, ,
            criteria=self.criteria)
        fill_kwargs = {"context": prompt, "llm": self.llm}
        if mode: fill_kwargs["mode"] = mode
        node = await ActionNode.from_pydantic(ReviewOp).fill(**fill_kwargs)
        response = node.instruct_content.model_dump()
        return response
class Revise(Operator):
    async def __call__(self, problem, solution, feedback, mode: str = None):
        prompt = REVISE_PROMPT.format(problem_description=problem, solution=solution, ,
            feedback=feedback)
        fill_kwargs = {"context": prompt, "llm": self.llm}
        if mode: fill_kwargs["mode"] = mode
        node = await ActionNode.from_pydantic(ReviseOp).fill(**fill_kwargs)
        response = node.instruct_content.model_dump()
        return response
```

B. Case Study

B.1. Case Study of DebFlow

```
SOLVE_PROMPT = """
Solve the given mathematical problem step by step. Show your work and explain each step
    clearly. If the problem involves geometry, include a description of the relevant
    geometric properties and relationships. If the problem is multiple choice, explain
    why the chosen answer is correct and why the others are incorrect.

Problem:
{input}

Provide a detailed solution:
"""
```

```
FORMAT_PROMPT = """
Format the given solution to match the following guidelines:
1. If there's a final numerical answer, enclose it in \boxed{}.
2. For multiple-choice questions, state the correct answer as a single letter (A, B, C,
    D, or E) without additional explanation.
3. If the answer is in radical form, leave it as is without simplifying to a decimal.
4. Ensure that all mathematical expressions are properly formatted using LaTeX notation
    where appropriate.

Given problem and solution:
{input}

Formatted answer:
"""

async def __call__(self, problem: str):
    """
    Implementation of the workflow
    """
    # Generate multiple solutions
    solutions = []
    for _ in range(3):  # Generate 3 solutions
        response = await self.custom(input=problem, instruction="")
        solutions.append(response['response'])

    # Review the generated solution
    reviewed_solution = await self.sc_ensemble(solutions=[solution], problem=problem)

    # Use the programmer to analyze and generate code for the reviewed solution
    code_solution = await self.programmer(problem=problem,
        analysis=reviewed_solution['response'])

    # Format the final answer
    formatted_answer = await self.custom(input=f"Problem: {problem}\nSolution:
        {code_solution['output']}", instruction=prompt_custom.FORMAT_PROMPT)

    return formatted_answer['response'], self.llm.cost_manager.total_cost
```

This optimal workflow generated for the MATH task showcases the model's ability to generate complex, task-specific solutions from task-agnostic initial settings. It combines programmatic solutions with various reasoning strategies, culminating in an ensemble selection process, and spontaneously formats the answer into the required form. This adaptation demonstrates the model's flexibility in tailoring workflows to different problem domains, while maintaining sophisticated problem-solving structures.

B.2. Case Study of AFlow

```
async def __call__(self, problem: str):
    """
    Implementation of the workflow
    """
    # Generate multiple solutions
    solutions = []
    for _ in range(3):  # Generate 3 solutions
        response = await self.custom(input=problem, instruction="")
        solutions.append(response['response'])

    # Review each generated solution for correctness
    reviewed_solutions = []
    for solution in solutions:
        review = await self.custom(input=solution, instruction="Review this solution for
            correctness.")
        reviewed_solutions.append(review['response'])
```

```
    # Use self-ensemble to select the best solution from reviewed solutions
    ensemble_response = await self.ensemble(solutions=reviewed_solutions, problem=problem)

    return ensemble_response['response'], self.llm.cost_manager.total_cost
```

When designing workflows, **Aflow** also generates multiple solutions. However, in the subsequent steps, it reviews each solution individually before integrating them, lacking a comprehensive analysis of how different solutions complement or contradict each other. In contrast, **Debflow** conducts an overall analysis before invoking `sc_ensemble()`, which enhances the consideration of the strengths and weaknesses of different solutions. This process enables `self.programmer()` to generate more reasonable code, whereas **Aflow** performs `ensemble()` only on the reviewed textual solutions, which may lead to information loss and a lack of validation at the code level. Additionally, **Debflow** further optimizes the final output using `self.custom()` in combination with `prompt_custom.FORMAT_PROMPT`, ensuring a clear and well-structured output. In contrast, **Aflow** lacks similar formatting mechanisms, making its final answer potentially less readable and structured.