Schema-Oriented Cognitive Processes for Expertise in Mathematics & Algorithms [SCOPE]

Mercy Dada modada@mit.edu

Nathaniel Morgan nmorgan@mit.edu

Noah Yared noah824@mit.edu

Samuel Manolis sammano@mit.edu

Abstract

In recent years, Large Language Models (LLMs) have demonstrated significant increases in their reasoning capabilities. Yet they frequently struggle to use consistent and accurate reasoning on both challenging and seemingly straight forward problems. While frameworks like Chain-of-Thought (CoT) encourage intermediate reasoning, they generally allow models free reign over the content of their thoughts rather than imposing structure on what the model should think about. Addressing this, we introduce Schema-Oriented Cognitive Processes for Expertise (SCOPE), a prompting approach that uses explicit, human-inspired algorithmic reasoning schemas to guide LLM reasoning. We evaluated SCOPE against Baseline, CoT, and ReAct strategies across eight algorithmic problem categories derived from the CLRS-text dataset. Our experiments with Qwen3-20B and Ministral3-14B demonstrate that SCOPE can provide gains to reasoning when the context and schema are carefully curated. However, the smallest model Qwen2.5-7B consistently performed worse in comparison to other reasoning approaches, suggesting that smaller models may struggle to effectively use schemas given that schemas significantly increase the context length.

Source code for SCOPE benchmarking is available at https://github.com/n-morgan/SCOPE-Benchmarking.

1 Introduction

With the emergence of reasoning in Large Language Models (LLMs), there has been no shortage of demand for utilizing their reasoning capabilities across various domains. Today, LLMs are increasingly becoming integrated into dynamic applications across diverse domains to complete tasks requiring reasoning over complex problems in pursuit of long-term goals. Additionally, users continue to expand their interactions with LLMs beyond simple text generation prompting and question answering to more agentic multi-turn interactions in pursuit of solving complicated and novel problems. Together, these trends drive the demand for ever improving robustness in LLM general reasoning capabilities. Advancements in post-training techniques, scaling quality training data, and increasing model sizes have contributed to drastic improvements in model reasoning compared to models from years prior. Currently, the LLM reasoning paradigm relies on using various techniques to direct the model to generate intermediary tokens (“thinking” tokens) before returning a final response to a prompt. Yet despite these advancements, LLMs are found to struggle with problem solving and reasoning consistency when tackling problems outside its trained distribution. Furthermore, even models capable of solving high-difficulty problems correctly show surprising failures when attempting to solve simpler problems that would be considered rudimentary to humans or when attempting to solve problems that exceed a certain difficulty threshold – problems that are essentially too difficult. A major theme underpinning these limitations is the issue that models can often be inconsistent, and faulty with their reasoning thoughts. Therefore, given these limitations we look to expand on the current LLM reasoning framework by not just encouraging models to “think” before producing an output, but by providing thought schemas to guide what models actually think about. In this paper, we explore Schema-Oriented Cognitive Processes for Expertise (SCOPE) as a prompting approach for guiding what LLMs think about when solving problems and examine it impact on model thinking consistency and accuracy.

2 Related Works

Kick-starting the development of reasoning in LLMs was the notion that LLMs might learn to reason by mimicking how humans break down problems into smaller intermediary steps then build to a final solution. The study by Wei et al. demonstrated that by providing sufficiently large LLMs with examples of human reasoning traces (chains-of-thoughts (CoTs)) in their prompt, such models were able to generate CoTs before arriving at an output. These examples were provided in the form of (input/prompt, chain-of-thoughts, output) triplets. Once prompted to generate CoTs, models were shown to produce more accurate outputs compared to the same model without CoT prompting. Building from this, studies looked to combine CoTs with actions allowing models to retrieve information or take action based on their thoughts (Yao et al. 2023). These combinations showed promising results towards reducing hallucinations produced by vanilla CoT, by grounding model outputs on the facts it retrieved. In further pursuit of generalized reasoning for problem solving, Yao et al. introduce a new paradigm whereby thoughts are no longer structured in a linear sequence. Instead, thoughts branch out to form a tree of thoughts (ToTs) that the model can then search over to find the correct output. Under this framework, LLMs showed improved accuracy on tasks requiring exploration, strategic look-ahead, or backtracking that were typically difficult with just CoT (Yao et al., 2023). Further advancements on the CoT framework continue to take inspiration from cognition literature and other machine learning techniques such as reinforcement learning to increase reasoning capabilities for problem solving.

However, current implementations of LLM reasoning still struggle with advanced problem solving. For example, Havrhilla et al. findings show that models struggle to reason beyond the reasoning patterns seen within the distribution of reasoning examples already seen. Furthermore, the study by Shojaee et al. finds that reasoning LLMs are limited in how well they can fix incorrect thoughts and generate novel thoughts. As a result, models fail to generate accurate outputs once a problem surpasses a threshold of complexity (Shojaee et al., 2025). When solving low complexity problems, LLMs often generate the correct solution early in their intermediate thoughts but continue to "overthink" by inefficiently exploring incorrect alternative thoughts. Such findings demonstrate limitations in the accuracy and consistency current LLM reasoning.

Despite this progress, the push in this domain towards improved reasoning primarily focuses on generalized reasoning implementations. Although some studies impose structures on the global reasoning process (e.g. ToTs), current implementations predominantly allow LLMs to have free reign over the content of their thoughts. There has been considerably less exploration on imposing structure over what exactly LLMs should think about. In this project, we aim to address this gap by investigating whether providing explicit algorithmic reasoning schemas (structured frameworks for individual thoughts) enhances the reasoning performance of LLMs. Here, we restrict our focus to mathematical and algorithmic domains in order to draw intuition from human reasoning methods similar to what has been done in prior research. Often when presented with mathematical or algorithmic problems, humans use pre-defined frameworks to reason through the problem. Using these frameworks enables us to constrain what information we deem relevant to solving the problem. Since LLMs have been shown to exhibit reasoning behavior when provided general human reasoning examples, similar methods could be employed to elicit models to use reasoning schemas within their thoughts. With this intuition, the use of reasoning schemas may address issues with model "overthinking" much like how cognitive schemas enable humans to reduce the scope of potentially relevant information. By providing explicit schemas, LLMs may be able to reason more efficiently as their use of schemas could decrease token expenditure on unnecessary thoughts, thus saving compute and resources. Potential implications of this project may also extend to applications where LLMs are required to go through repeated reasoning steps. In such scenarios, LLMs equipped with schemas may be able to reason with increased accuracy and consistency on these tasks.

Drawing from these motivations, our project seeks to explore how providing LLMs with explicit algorithmic reasoning schemas inspired by human problem-solving methods impacts reasoning accuracy compared to traditional unstructured reasoning methods built on chain-of-thoughts. We ask whether providing schemas explicitly can make LLM reasoning more accurate, consistent, and interpretable.

3 Method

Schema Design

To approach this question we utilized the CLRS-text dataset generator together with the pregenerated tomg-group-umd/CLRS-Text-train split. This dataset spans over 30 different algorithmic problems types, which we group together into the following categories: Sorting, Searching, Divide and Conquer, Greedy, Dynamic Programming, Graphs, Strings, Geometry. Each category is accompanied with a unique schema that details generalized thinking strategies specific to the problems within the category. These schemas are then coupled with the query, instructions on how to use the schema, and a worked out example problem using the schema to create the final prompt. We use these four elements in the input to prompt the model to use our designed schemas in its own reasoning.

To craft each category's general schema we sampled a set of problems from each category and examined each problem for common cognitive processes, strategies, and variables one would need to use or track in order to solve the problem. Using the common traits among sampled problems, we create a schema for each category requiring the model to keep track and detail the problem format, key variables, its progress through the problem, and the strategy it employs as it solves the problem. In the following section, we describe in further detail the key characteristics and intuition behind said characteristics for each problem category's schema. See appendix for detailed descriptions of each schema.

Sorting: We designed the generalized sorting schema based on specific examples of insertion sort, bubble sort, heap sort, and quick sort from the CLRS-text dataset, as well as prior knowledge of additional sorting algorithms such as merge sort and selection sort. We framed the overall design of the schema with the idea that at each step towards solving a sorting problem, the model should work through the schema to think through its progress and next steps. From our analysis of specific sorting algorithms, we determine four thought elements for the schema: current step representation, core state representation, focus action, and general iteration process. For the current step element, we determined that keeping tracking of the number of steps is important for knowing when to terminate the sorting loop. The current state representation element was designed using insights from examples of insertion, bubble, heap, and quick sorting problems, given that the CLRS-text dataset only contains examples of these algorithms. Here we found common concepts across sorting problems were to understand the input's representation; then with each step track the current representation of the modified input, the region of the input that is sorted, and region of the input that is left to be sorted. We determined these traits to be important for understanding what step to take next given the progress the algorithm already made. For the focus action element, we determined a common concept to take the next sorting step in each algorithm was to track the current element being sorted and the type of comparisons being made with other elements. Lastly we provide a generalized iteration loop to guide looping through each sorting step.

Searching: We designed a schema that solves three classic array-based searching problems: locating a target in a sorted array with binary search, finding the minimum element by linear scan, and selecting the k-th smallest element with quickselect. Inputs provide a numeric list key, an optional scalar target for binary search or rank k for quickselect, and an initial_trace pair of indices (a, b) specifying the initial active interval. For binary search and quickselect, the schema tracks the inclusive index interval (low, high) that represents the current window of the array it is considering, while for minimum finding it tracks the scan position (i, end) together with the running minimum index. At each iteration the algorithm updates these control variables according to its usual logic, shrinking the interval to the half that can still contain the target for binary search or narrowing the quickselect subarray around a pivot, and appends the new pair to the trace. It then returns the terminal pair in the form (x_f, y_f), where the final state encodes either the exact index (e.g., (i, i) for a found element or minimum or the collapsed interval where the search converged.

Divide and Conquer: We structured the divide-and-conquer schema around four components: *base case*, *divide*, *conquer*, and *combine*. In *base case*, the model directly computes the answer for any input of size at most 1. In *divide*, we instruct the model to define a mapping `F` that takes a problem instance `I` with `|I| > 1` and returns the set of subproblems `S` derived from `I` according to the algorithm. Applying `F` to the original problem `P_0` yields the initial set of subproblems `S_0 = F(P_0)`. In *conquer*, we recursively apply `F` to the subproblems in `S_0` until each of our subproblems is a base case. This process constructs a recursion tree `T`, where the nodes of `T` represent subproblem inputs and satisfy that each child node `I_c` of the parent `I_p` is contained in `F(I_p)`; the leaves `I_leaf` satisfy `|I_leaf| <= 1`. In *combine*, we define a mapping `G`, that builds the answer to a given problem `P` from the outputs of the set of problems `F(P)`. Letting `H = height(T)`, we solve subproblems bottom-up: at each level `h`, the result for any non-leaf node `I` is

$$\texttt{O(I) = G(\{O(c): c} \in \texttt{children(I)\})}.$$

When we reach the root node `P_0` at height `H`, we return the final result

$$\texttt{O(P\_0) = G(\{O(s): s} \in \texttt{S\_0\})}.$$

Greedy: We designed the greedy schema as a four-step template mirroring the greedy-choice paradigm. First, we initialize any problem-specific state or variables, such as setting `last_finish = -inf` in the activity-selection problem. Second, we define a greedy criterion: a mapping `greedy_key` from input items to $\mathbb{R}$ that determines an order in which items should be considered to construct an optimal greedy solution. The model is instructed to sort the input items according to this key. Third, we specify a feasibility predicate `feasible(item, selected)`, where `item` is the input item being tested for feasibility and `selected` is the current selected set of items; it should return true if adding `item` to `selected` does not violate problem constraints. In activity selection, for example, this predicate checks that the activity `item`'s interval does not overlap with any of the activities in `selected`. Finally, we guide the model to traverse the sorted items and add each item to the solution set `selected` if the feasibility predicate is satisfied, incrementally building up the optimal greedy solution.

Dynamic Programming: We designed the generalized dynamic programming (DP) schema, using the SRTBOT framework. The SRTBOT framework solves DP problems by defining the following: S - the subproblems the problem can be recursively broken down into; R - the recurrence relations between subproblem; T - the topological order of subproblems ensuring that the global problem can be solved from the combination of subproblems; B - the base cases for the subproblems; O - the original problem; and T - the time complexity/analysis for this framework to solve the original problem. These components cover the key general components necessary to solve DP problems presented in Introduction to Algorithms by Cormen et al., from which the CLRS-text dataset is derived. Using the SRTBOT framework we developed the DP schema to contain four thought elements: current step representation, current state representation, focus action, and general iteration process.
For the current step representation, we include elements to help the model keep track of its current step in the subproblem space. Here we track the current number of steps and the current subproblem. For the core state representation, we encode the SRTBOT framework above by having the model track the input representation, the overall subproblem table mapping the relationship between subproblems, the current subproblem, the base cases and termination condition. These map to tracking the subproblem, topological order, base cases, and original problem components from the SRTBOT framework. We designed the focus action element to contain concepts for the recurrence relation and the chosen values for subproblems along the way. Lastly we also provide a general iteration process detailing the recursive nature to solving the subproblems.

Graphs: The graphs schema was designed to encompass the variety of questions under the graph category while maintaining strong guidelines for each. After isolating the key concepts of each algorithm and the inputs and outputs expected, LLM assistance was used to find overlaps between each algorithm. These overlaps helped create a concise schema which minimized redundancy while ensuring clarity across each question.

Strings: The strings schema was tailored to the two algorithm types under the strings category. The schema includes general outlines for how to solve a question of either problem type. After isolating the key details of each algorithm and the inputs and outputs expected, LLM assistance was used to stitch information into a reproducible schema.

Geometry: We designed a unified geometry schema that solve the three classic computational geometry problems provided in the dataset: detecting whether two line segments intersect and computing planar convex hulls via Graham scan and Jarvis' March. Inputs provide point sets as parallel arrays x and y with fixed indices, and convex hull problems include an `initial_trace` zero vector simply indicates the shape of the expected 0/1 outputs. The schema first determines the orientation of the provided arrays via a signed cross product that reports whether three points turn left, right, or are collinear. For hulls, we use a consistent tie rule that selects the farthest point when multiple points lie on the same ray. In Graham

Scan problems, we maintain a stack of candidate hull vertices and record how this list changes by emitting length-n 0/1 snapshots after each vertex removal (pop) and addition (push). In Jarvis' March, we instead track the growing hull set and record a snapshot each time a new vertex is wrapped onto the hull. Both hull methods output a the final 0/1 vector that marks exactly which points belong to the completed hull. The segment intersection task takes two segments defined by four coordinates, applies the orientation and on-segment checks, and outputs a single numeric label 0 or 1 indicating whether the segments are disjoint or intersecting.

Implementation

For each specific problem under a given category we prompt the model by providing a specific instruction, a worked example using the categories schema, the problem prompt, and a blank schema using the following format:

- Instruction: "You are a problem-solving agent capable of..."
- Two solved examples from the same category
- Prompt: "Now answer the following question {question} using the schema below. Enclose your answer in </answer> your answer <answer>"
- One blank schema corresponding to the question category's schema
- Answer: {answer}

COT Dataset Creation For each category:

- Instruction: "You are a problem-solving agent capable of..."
- Two general COT examples
- Prompt: "Now answer the following question {question} using the schema below. Enclose your answer in </answer> your answer <answer>"
- One blank COT schema
- Answer: {answer}

ReAct Dataset Creation For each category:

- Instruction: "You are a problem-solving agent capable of..."
- Two general ReAct examples
- Prompt: "Now answer the following question {question} using the schema below. Enclose your answer in </answer> your answer <answer>"
- One blank ReAct schema
- Answer: {answer}

Control Dataset Creation For each category:

- Instruction: "You are a problem-solving agent capable of..."
- Prompt: "Now answer the following question {question} using the schema below. Enclose your answer in </answer> your answer <answer>"
- Answer: {answer}

4 Experiments

We evaluate all prompting strategies all eight algorithmic problem categories. All experiments were run on a workstation equipped with two NVIDIA RTX 3090 GPUs, using Python-based evaluation scripts for exact-match scoring. Our study examined whether imposing explicit structure on model reasoning through problem-specific schemas improves performance on algorithmic tasks relative to general prompting strategies. We performed a comparative evaluation across four prompting methods: Baseline, Chain of Thought, ReAct, and our schema-guided SCOPE framework. Three contemporary open-source models were assessed: Qwen2.5-7B, Qwen3-20B, and Ministral3-14B.

Exact-match accuracy was measured across eight algorithmic categories to enable a fine-grained comparison of how structured reasoning affects model behavior. Improvements achieved by SCOPE over Chain of Thought, ReAct, and the Baseline were interpreted as evidence that explicit schema guidance enhances the consistency and correctness of algorithmic reasoning. Cases in which SCOPE matched or underperformed these methods indicated that schema-based reasoning does not universally provide an advantage under this evaluation setting.

Our primary evaluation metric is final answer accuracy, computed via exact string matching. This metric provides a direct means of comparing correctness across prompting methods with differing reasoning structures and aligns with the metric used by Markeeva et al. in their evaluation of algorithmic reasoning models. For benchmarking, we use the tomg-group-umd/CLRS-Text-train dataset and evaluate each prompting strategy on 1000 randomly sampled instances, corresponding to approximately 125 problems per category. This dataset is generated by the CLRS-Text problem generator employed by Markeeva et al., making it well suited for measuring reasoning capabilities across a wide range of algorithmic tasks.

As described above, problems fall into eight categories: sorting, searching, divide and conquer, greedy, dynamic programming, graphs, strings, and geometry. This breakdown allows us to assess how schema-based reasoning scales across diverse algorithmic paradigms while maintaining consistent task complexity and restricting attention to polynomial-time problems. The same dataset is used for both domain-specific and general prompting strategies to enable direct comparison.

5 Results

In this section, we focus on final-answer accuracy as our primary metric for comparing methods and leave measuring cross-run consistency for future work (see Discussion & Future Steps).

Model	Baseline	CoT	ReACT	SCOPE
Qwen2.5 7B	12.73%	**20.37%**	19.37%	14.43%
Qwen3 20B	16.53%	20.87%	22.93%	**26.50%**
Ministral3 14B	15.67%	12.00%	18.67%	**21.67%**

Table 1: Exact match accuracy across prompting strategies for three model families. Best performance per model family is shown in bold.

Baseline	Qwen2.5 7B	Qwen3 20B	Ministral3 14B
Divide & Conq	0/100	5/100	0/10
Dynamic Prog	0/300	3/300	0/30
Geometry	111/300	115/300	0/30
Graphs	123/1200	201/1200	9/120
Greedy	16/200	22/200	0/20
Searching	4/300	32/300	0/30
Sorting	104/400	86/400	38/40
Strings	24/200	32/200	0/20

Table 2: Baseline performance across all models and categories.

CoT	Qwen2.5 7B	Qwen3 20B	Ministral3 14B
Divide & Conq	10/100	12/100	0/10
Dynamic Prog	1/300	4/300	1/30
Geometry	92/300	126/300	1/30
Graphs	184/1200	246/1200	16/120
Greedy	24/200	24/200	4/20
Searching	55/300	77/300	0/30
Sorting	209/400	102/400	14/40
Strings	36/200	35/200	0/20

Table 3: CoT performance across all models and categories.

ReACT	Qwen2.5 7B	Qwen3 20B	Ministral3 14B
Divide & Conq	6/100	13/100	1/10
Dynamic Prog	5/300	6/300	0/30
Geometry	107/300	132/300	1/30
Graphs	178/1200	245/1200	21/120
Greedy	16/200	22/200	0/20
Searching	14/300	75/300	1/30
Sorting	223/400	153/400	32/40
Strings	32/200	42/200	0/20

Table 4: ReACT performance across all models and categories.

SCOPE	Qwen2.5 7B	Qwen3 20B	Ministral3 14B
Divide & Conq	0/100	0/100	6/10
Dynamic Prog	6/300	4/300	0/30
Geometry	67/300	111/300	11/30
Graphs	93/1200	194/1200	19/120
Greedy	0/200	33/200	1/20
Searching	20/300	55/300	7/30
Sorting	234/400	340/400	19/40
Strings	13/200	58/200	2/20

Table 5: SCOPE performance across all models and categories.

5.1 Table Summary

From Table 1, we observe that SCOPE outperformed all three comparison methods on the two largest models (Qwen3-20b and Ministral3-14b), exceeding the second-best exact match accuracy by more than 15%. In contrast, on the smallest model (Qwen2.5-7b), SCOPE surpassed the baseline by around 15% but performed over 25% worse than CoT and ReAct.

Tables 2-5 report per-category performance for each model across the four prompting strategies. For Qwen2.5-7b, the baseline performed at least as well as SCOPE in more than half of the problem categories, indicating little benefit from our schema-based approach. For Qwen3-20b, SCOPE notably outperformed the other three methods in Sorting, increasing accuracy from 38% with ReAct to 85% and performed comparably in the other categories. Despite struggling with Divide & Conquer on the previous two models, SCOPE achieved a significant improvement over the other three methods on Ministral3-14b, increasing accuracy from 10% with ReAct to 60%. Furthermore, on Ministral3-14b, SCOPE exceeded the baseline in every category except Sorting.

5.2 Analysis

These results suggest that SCOPE is more effective on larger models, outperforming or matching standard prompting strategies such as ReAct and CoT, while its effectiveness diminishes on smaller models. We hypothesize that this behavior arises primarily from two factors: template complexity and attention dilution.

5.2.1 Template Complexity

According to a study by Tang et al. (2025), task difficulty is more sensitive to task depth than to task width in LLM single-agent systems. The depth of a task is the length of the reasoning chain (the number of sequential problem-solving steps) that the agent must traverse, whereas the width denotes the diversity of capabilities required for the task. This is primarily a result of error accumulation, where errors compound over long reasoning chains, causing model performance to degrade exponentially.

Therefore, using task depth as a proxy for the difficulty of problem categories in our experiment, we find that dynamic programming, divide and conquer, and graphs provide the largest challenges. Problems within these categories require recursively solving nested subproblems, whose results are combined to build the full solution. On the other hand, problems in categories such as sorting, strings, or searching mostly rely on localized, independent reasoning steps, which mainly involve pairwise element comparisons. As a result, these problem tasks are comparatively shallow and narrow in the diversity of capabilities they require.

Due to the inherent recursive nature of the problems, we designed the Divide & Conquer and Dynamic Programming schemas to be recursive as well; that is, the LLM was forced to apply the schemas recursively to algorithmic tasks. However, for the other categories, schemas were mostly iterative. Thus, our Divide and Conquer and Dynamic Programming schemas were likely the most complex templates for models to follow. For clarity, template complexity refers to the structured burden imposed by SCOPE, whereas task complexity is inherent to the problem category. In this case, however, they align, since recursive tasks require recursive schemas.

Shojaee et al. (2025) show that the task-solving accuracy of frontier LLMs (both thinking and non-thinking) completely collapses once a certain task complexity threshold is reached. Even when provided with an explicit algorithm for the task, models fail to reason consistently and apply the algorithm precisely. These findings help explain the generally unremarkable performance of SCOPE in problem categories like Divide & Conquer and Dynamic Programming across the three models. The added reasoning overhead and token cost for filling out the schemas likely outweighed their benefits, given the limited ability of LLMs to consistently follow a specified sequence of reasoning steps.

5.2.2 Attention Dilution

We measured the token lengths of the prompt templates for each method, as well as the actual questions from the CLRS-Text dataset, so that we could conduct a more thorough investigation of our experimental results. To do this, we used the OpenAI GPT-2 tokenizer from the HuggingFace transformers library. The measurements for the full prompt templates (excluding the question) were: 124 tokens for baseline, 412 tokens for CoT, 496 tokens for ReAct, and $\sim$2090 tokens for SCOPE (averaged across all the problem categories). Moreover, we measured the CLRS-Text questions (post-processing) to be ~ 94.1 tokens long on average. From the above token statistics, the average ratio of the number of prompt template tokens that were not part of the question to the number of question tokens for SCOPE was ~ 22.2, whereas the ratios for baseline, CoT, and ReAct were 1.32, 4.38, and 5.27, respectively (more than four times smaller than SCOPE).

These numbers suggest that attention dilution may have negatively affected how SCOPE performed compared to the other three methods. Due to the high ratio of tokens not directly relevant to the question in the SCOPE prompts, the LLMs may have struggled to attend to useful or relevant tokens within the schemas and few-shot examples we provided. Because of transformers' fixed attention budget, model attention to tokens relevant to solving the question may have been diluted, resulting in models ineffectively applying the schemas. On the other hand, with the other three methods, the question tokens, which were densely packed and highly relevant to the tasks, made up a larger portion of the prompts, comparatively mitigating the effects of attention dilution.

6 Discussion & Future Steps

These findings suggest that while SCOPE has potential to boost reasoning accuracy and consistency, these gains were experienced in more nuanced settings. Overall, we find SCOPE to be more effective as the model size scales, which we attribute to be a result of large models having stronger capabilities to digest large contexts without losing precision. With our designed schemas containing highly detailed components, precise processing of the schema is necessary to use it consistently. Further expanding on the importance of precision, we find that in our problem categories where the query problem was well defined, models were able to more effectively use schemas to make sense of the problem. Adding hand-crafted schemas to already weakly defined or understandable problems seemed to present more opportunities for the models to apply schema components to incorrect parts of the problem or simply get confused. In addition, performance variation across the eight problem categories even for models where SCOPE outperformed overall indicates that careful design of schemas with respect to their size and granularity within problem domains is necessary to capture gains. For problems that are concretely described, using direct schemas generally appears to boost performance. However, more abstract problem spaces may require less direct schemas of different forms.

One significant limitation to our study arises from this concept of schema design. Given that our schemas were hand crafted we adopted an iterative approach to the schema design after identifying key thought components. This led to benchmarking only on the final iteration of the schema. Therefore aside from design differences between schemas for different categories our method does not dive further into the impact of design choices within a problem category. Another limitation arises from our prompting strategy. Although we designed our prompts to encourage schema use, such an approach does not guarantee that the model uses the thought schema at each step of its reasoning. From some initial trace inspections, we observed that the models might only use the schema for one step then continue to reasoning without it. Further, with our evaluations we did not fully investigate model traces which would offer more detailed insights into how models are making use of the schemas, which would allow us to more deeply assess SCOPE's effectiveness in improving both accuracy and consistency. We also note limitations from benchmarking on a small set of models. While performance results proved promising, the number of models we tested limits our ability to concretely generalize SCOPE to all LLMs. Going forward, testing SCOPE with multiple models from different size categories, small, medium, large, may offer deeper insights to which models benefit the most from SCOPE.

Future research may focus on investigating how various design choices for schemas, such as the length and granularity of a schema affect model performance. Additionally, we see further room for exploration on the effects of model size on performance using SCOPE. We also acknowledge that hand-crafting these schemas is counter to the push for increasingly allowing components to be learned from data. Therefore, we see significant potential in exploring the use of schemas but under the framework that schemas are designed or learned from training rather than hand-crafted by humans.

References

Alex Havrilla, Yuqing Du, Sharath Chandra Raparthy, Christoforos Nalmpantis, Jane Dwivedi-Yu, Maksym , Zhuravinskyi, Eric Hambro, Sainbayar Sukhbaatar, and Roberta Raileanu. 2024. Teaching large language models to reason with reinforcement learning. *arXiv preprint arXiv:2403.04642*.

Larisa Markeeva, Sean McLeish, Borja Ibarz, Wilfried Bounsi, Olga Kozlova, Alex Vitvitskyi, Charles Blundell, Tom Goldstein, Avi Schwarzschild, and Petar Veličković. 2024. The clrs-text algorithmic reasoning language benchmark. *arXiv preprint arXiv:2406.04229*.

P. Shojaee, I. Mirzadeh, A. Keivan, H. Maxwell, B. Samy, and F. Mehrdad. 2025. The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity.

B. Tang, H. Liang, K. Jiang, and X. Dong. 2025. On the importance of task complexity in evaluating llm-based multi-agent systems. *arXiv preprint arXiv:2510.04311*.

Petar Veličković, Adrià Puigdomènech Badia, David Budden, Razvan Pascanu, Andrea Banino, Misha Dashevskiy, Raia Hadsell, and Charles Blundell. 2022. The clrs algorithmic reasoning benchmark. *arXiv preprint arXiv:2205.15659*.

J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*.

S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan. 2023a. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*.

S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. 2023b. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

A Appendix

Schemas

A.1 Graphs

Input

- **algorithm:** <algorithm_name> (e.g., DFS, BFS, Dijkstra)
- **graph:**
  - **type:** "adjacency_list" or "adjacency_matrix"
  - **value:** [...]
- **source_node:** <s> (only for single-source algorithms)
- **example_output:** [...] (optional)

Step 1: Initialize variables

DFS / BFS
- `pi = [NIL for each node]` (predecessor array)
- `visited = [False for each node]`

Dijkstra / Prim
- `pi = [NIL for each node]`
- `dist = [`∞` for each node]` (Dijkstra)
- `key = [`∞` for each node]` (Prim)
- `visited = [False for each node]`

Topological Sort
- `visited = [False for each node]`
- `topo_order = []`

Articulation Points / Bridges / SCC
- `discovery_time = [NIL for each node]`
- `low = [NIL for each node]`
- `pi = [NIL for each node]`
- `visited = [False for each node]`

Step 2: Select traversal / algorithm

- DFS: recursive or stack-based deep traversal
- BFS: queue-based level traversal
- Topological Sort: DFS with post-order appends
- Articulation Points / Bridges: DFS with low values
- SCC (Kosaraju): two-pass DFS (first for finishing times, second on transposed graph)
- Dijkstra: min-heap, extract-min and relax neighbors
- Prim: min-heap, extract-min key and update neighboring keys

Step 3: Iterate through nodes / edges

- DFS / BFS / Topological / SCC / AP/Bridges:

```
for each node u in graph:
    if not visited[u]:
        call dfs_visit(u)
```

- Dijkstra / Prim:

```
while priority_queue not empty:
    u = extract_min()
    for each neighbor v of u:
        relax(u, v)
```

Step 4: Update final output

- DFS / BFS: `pi` array (predecessor tree)
- Topological Sort: `topo_order` array
- Articulation Points / Bridges: list of nodes or edges
- SCC (Kosaraju / Tarjan): component assignment per node
- Dijkstra / Prim: `pi` array + `dist/key` array

Step 5: Solve for final output

```
Output:
{
  "pi": [...],
  "dist": [...],
  "key": [...],
  "topo_order": [...],
  "components": [...]
}
```

A.2 Strings

Input

- **algorithm:** `<algorithm_name>` ("Naive" or "KMP")
- **string:** list of symbols representing the main text
- **pattern:** only for Naive
- **key:** prefix-function for KMP (only for KMP)
- **example_output:** optional, default `None`

Step 1: Initialize State

state initialize as empty dictionary
first_match stores the first match index (initially `None`)

- **Naive:**
  - `string_length = len(string)`
  - `pattern_length = len(pattern)`
- **KMP:**
  - `pi = key` (prefix-function array)
  - `current_index = 0` (index in string)
  - `pattern_index = 0` (index in pattern)
  - `string_length = len(string)`
  - `pattern_length = len(key)`

Step 2: Preprocess Pattern

- Naive: no preprocessing required
- KMP: key (prefix-function) is already provided

Step 3: Search / Iterate Through String

Naive:
```
for s in range(string_length - pattern_length + 1):
    match = True
    for i in range(pattern_length):
        if string[s + i] != pattern[i]:
            match = False
            break
    if match:
        first_match = s
        break
```

KMP:
```
for current_index in range(string_length):
    while pattern_index > 0 and string[current_index] != string[pattern_index]:
        pattern_index = pi[pattern_index - 1]
    if string[current_index] == string[pattern_index]:
        pattern_index += 1
    if pattern_index == pattern_length:
        first_match = current_index - pattern_length + 1
        break
```

Step 4: Return Final Output

- `output = first_match`
  Returns the index of the first match, or `None` if no match is found

A.3 Sorting

Input

- **algorithm:** <sorting_strategy> (e.g., Insertion, Bubble, Heap, Quick)
- **input_list:** list of comparable elements (e.g., [5, 3, 1, 4])
- **example_output:** optional

Step 1: Initialize State Representation

Input Representation
- state["input"] = current snapshot of the list
  - type: list[Any]

Current Stage
- tracks internal progress (heapified array, partition boundaries, etc.)
  - type: list[Any] or dict[str, Any]

Boundary of Sorted Region
- tracks which indices are sorted
  - type: list[int] or tuple[int, int]

Remaining
- tracks indices still needing sorting
  - type: list[int] or tuple[int, int]

Termination Condition
- boolean or text condition
  - example: "remaining is empty"

Step 2: Select Focused Action

- **Current Element Focus:** element currently being compared / moved
- **Comparison Targets:** elements or regions used for comparison
- **Operation Type:** "compare", "swap", "insert", "partition"
- **Reasoning / Action Plan:** rule such as
  "swap if current < target" or "move pivot to boundary"

Step 3: Core Iteration Loop

```
for i in range(len(input_list)):
    current = input_list[i]            # Current Element Focus
    comparison_targets = input_list[:i]# Comparison Targets
    operation_type = "compare"

    # Reasoning:
    # If current element is smaller than any earlier element,
    # move it backward like insertion sort.
    for j in range(i - 1, -1, -1):
        if input_list[j] > input_list[j + 1]:
            # Swap
            input_list[j], input_list[j + 1] = \
                input_list[j + 1], input_list[j]
            operation_type = "swap"
        else:
            break
```

Step 4: Final Output

- sorted_list = input_list
- num_operations = ... (optional)
- trace = ... (optional reasoning trace)

A.4 Dynamic Programming

Input

- **problem_name:** `<problem>`
- **input_representation:** snapshot of the problem inputs
- **example_output:** optional

Step 1: Step Metadata

Step ID: None
Problem Name: ""
Operation Type: ""
Explanation: ""

State Representation

Input Representation • type: `Any | list[Any] | dict[str, Any]`
- description: Snapshot of the problem inputs.

Subproblem Table • type: `list[Any] | dict[tuple, Any]`
- description: Stores solutions to subproblems (e.g., `dp[i][j]`, `memo[(i,j)]`).

Current Subproblem Focus • type: `tuple[int,...] | str`
- description: Subproblem currently being computed or considered.

Base Case • type: `l`
- description: When the subproblem should return a direct value and which value.

Termination Condition • type: `bool | str`
- description: Condition signaling completion (e.g., “all subproblems computed”).

Step 2: Focused Action

Recurrence Application • type: `Any`
- description: Result of combining previously computed subproblems according to recurrence.

Decision Variables / Choices • type: `Any | list[Any] | tuple`
- description: Decision variables used in recurrence.

Reasoning / Action Plan • type: `str`
- description: Describes computation logic and intended updates to the DP table.

Step 3: Core Iteration Process

```
dp_table = {}
subproblem_order = []
termination_condition = False
step_id = 0

while not termination_condition:
    current_subproblem = f"Subproblem_{step_id}"
    subproblem_order.append(current_subproblem)

    is_base_case = (step_id == 0)

    if is_base_case:
```

```
            dp_table[current_subproblem] = ""   # Base case placeholder
            operation_type = "base case"
            reasoning = f"Identified {current_subproblem} as a base case"
        else:
            dependent_subproblems = list(dp_table.keys())
            dependent_values = [dp_table[sub] for sub in dependent_subproblems]

            dp_table[current_subproblem] =
                sum(dependent_values) + recurrence_value

            operation_type = "compute recurrence"
            reasoning = (
                f"Computed {current_subproblem} using values from "
                f"{dependent_subproblems} → {dp_table[current_subproblem]}"
            )

        step_id += 1

        # Termination logic:
        if termination_conditions:
            break
```

Step 4: Output

- **dp_table:** stores results of all computed subproblems
- **subproblem_order:** order in which subproblems were computed

A.5 Geometry

Input

- **algorithm:** <algorithm_name> (e.g., "segments_intersect", "graham_scan", "jarvis_march")
- **x:** list of x-coordinates for each point (e.g., [x_0, x_1, . . . , x_n-1])
- **y:** list of y-coordinates for each point (e.g., [y_0, y_1, . . . , y_n-1])
- **example_output:** optional reference output (e.g., intersection label or hull indicator array)

Step 1: Preprocessing

- Form point list P = [(x[i], y[i]) for i in range(n)].
- For "segments_intersect":
  - Identify the endpoints of the two line segments from P.
- For "graham_scan" and "jarvis_march":
  - Identify an anchor point (lowest y coordinate; break ties by lowest x).

Step 2: Select Geometry Procedure

- **segments_intersect:**
  - Use orientation tests and bounding box checks to determine whether the two segments intersect.
- **graham_scan:**
  - Sort all points by polar angle with respect to the anchor.
  - Maintain a stack or list for hull construction using left-turn tests.
- **jarvis_march:**
  - Initialize the hull with the anchor point.
  - Repeatedly choose the most counterclockwise point relative to the last hull point.

Step 3: Iterate Through Elements

- **segments_intersect:**
  - Evaluate orientation tests orient(a, b, c) for the required endpoint combinations.
  - Combine orientation results with bounding box checks to decide if the segments intersect.
- **graham_scan:**

```
for each point p in points_sorted_by_polar_angle:
    while len(hull) >= 2 and last_turn_is_not_left(hull[-2], hull[-1], p):
        hull.pop()
    hull.append(p)
```

- **jarvis_march:**

```
hull = [anchor]
current = anchor
while True:
    next_point = any_other_point
    for p in all_points:
        if p is more_counterclockwise_than(next_point, current):
            next_point = p
    current = next_point
    if current == anchor:
        break
    hull.append(current)
```

Step 4: Final Output

- **segments_intersect:**
  - Output `1` if the two segments intersect, otherwise `0`.
- **graham_scan:**
  - Output a binary array of length `n` marking hull vertices with `1` and non-hull points with `0`.
- **jarvis_march:**
  - Output a binary array of length `n` marking hull vertices with `1` and non-hull points with `0`.

Output (Schema-Filled Representation)

```
{
  "answer": ...   # integer (0/1) for intersection,
                  # or binary list of length n for hull algorithms
}
```

A.6 Searching and Selection

Input and General Conventions

- Inputs come from a question object:
  - `question.key` list of numbers (the array)
  - `question.target` number, only for `binary_search`
  - `question.k` integer, 0-based rank, only for `quicksearch`
  - `question.initial_trace` tuple of two indices `(a, b)`, inclusive
- **algo_name** selects which procedure to run:
  - `"binary_search"`
  - `"minimum_finding"`
  - `"quicksearch"`
- We maintain and record a **state trace** of the main control variables per step.
- Trace format to emit: `"(x1, y1), (x2, y2), ... | (xf, yf)"`
  Everything before `"|"` are intermediate states. The pair after `"|"` is the terminal state.

Dispatcher (Conceptual)

```
if algo_name == "binary_search":
    run BINARY_SEARCH with:
        key = question.key
        target = question.target
        (low, high) = question.initial_trace
elif algo_name == "minimum_finding":
    run MINIMUM_FINDING with:
        key = question.key
        (start, end) = question.initial_trace
elif algo_name == "quicksearch":
    run QUICKSEARCH with:
        key = question.key
        k = question.k
        (low, high) = question.initial_trace
```

Step 1: Binary Search

Goal

Given a sorted array `key` and a `target`, narrow the interval `[low, high]` until the search converges or finds an exact match. Record the interval after each update.

Inputs

- **key:** sorted ascending list of numbers
- **target:** number to search for
- **(low, high):** inclusive indices, from `question.initial_trace`

State and Invariant

- Maintain `low` and `high` as inclusive bounds on the candidate region.
- Invariant: all possible indices of the answer remain within `[low, high]` at each step.

Pseudocode

```
trace = []                    # list of (lo, hi)
lo = low
hi = high
```

```
while lo <= hi:
    mid = floor((lo + hi) / 2)
    if key[mid] == target:
        lo = mid
        hi = mid
        trace.append((lo, hi))
        break
    elif key[mid] < target:
        lo = mid + 1         # discard left half including mid
        trace.append((lo, hi))
    else:
        hi = mid - 1         # discard right half including mid
        trace.append((lo, hi))

# Final state:
# If target found, terminal state is (mid, mid).
# If not found, loop exits with lo = insertion point and hi = lo - 1.
```

Final State and Output

- If `target` was found, the terminal pair is `(mid, mid)`.
- If not found, the loop exits with `lo` as the insertion point and `hi = lo - 1`. The terminal pair can be taken as `(lo, hi)` (or clamped if desired).
- **Output pair:** `(lo, hi)`

Step 2: Minimum Finding (Linear Scan)

Goal

Find the index of the minimum element in key within `[start, end]`. Record the active scan window as it progresses.

Inputs

- **key:** list of numbers
- **(start, end):** inclusive scan bounds, from `question.initial_trace`

State and Invariant

- Index `i` moves from `start` to end, inclusive.
- `min_idx` holds the index of the smallest value seen so far within `[start, i]`.
- The scan window after processing position `i` can be viewed as the pair `(i, end)`.

Pseudocode

```
trace = []
i = start
j = end
min_idx = start

for pos in range(i, j + 1):
    if key[pos] < key[min_idx]:
        min_idx = pos
    trace.append((pos, j))   # record progress of the scan

# Final state:
# Terminal pair is (min_idx, min_idx).
```

Final State and Output

- **Terminal pair:** `(min_idx, min_idx)`
- **Output pair:** `(min_idx, min_idx)`

Step 3: Quicksearch (Quickselect for k-th Order Statistic)

Goal

Return the `k`-th smallest element (0-based) from `key` within the current subarray. Record the active subarray bounds `(lo, hi)` at each narrowing step.

Inputs

- **key:** list of numbers
- **k:** integer, 0-based rank to select
- **(low, high):** inclusive subarray bounds to search
- **pivot rule (optional):** `"first"`, `"last"`, or `"median_of_three"`

State and Invariant

- Maintain `lo` and `hi` as inclusive bounds on the subarray where the `k`-th element lies.
- At each step, partition around a pivot. After partition:
  - All elements left of `pivot_pos` are less than the pivot value.
  - All elements right of `pivot_pos` are greater than or equal to the pivot value.
- Invariant: `k` remains within `[lo, hi]` and the sought element lies in that subarray.

Pseudocode

```
trace = []
lo = low
hi = high

while lo <= hi:
    trace.append((lo, hi))

    # choose pivot index p_idx using a simple rule:
    #   first:  p_idx = lo
    #   last:   p_idx = hi
    #   median_of_three: p_idx = median_index_of(lo, mid, hi)

    pivot_pos = PARTITION(key, lo, hi, p_idx)

    if pivot_pos == k:
        terminal = (k, k)    # found exact rank
        break
    elif pivot_pos < k:
        lo = pivot_pos + 1   # search right side
    else:
        hi = pivot_pos - 1   # search left side

# Final state:
# If pivot_pos == k, terminal is (k, k).
# Otherwise terminal is the collapsed interval (lo, hi) after the loop.
```

Final State and Output

- If `pivot_pos == k`, the terminal pair is `(k, k)`.

- Otherwise, the terminal pair is the final collapsed interval `(lo, hi)`.
- **Output pair:** `(lo, hi)`

Partition Procedure (Lomuto-Style, Index-Based)

```
PARTITION(A, lo, hi, p):
    pivot = A[p]
    swap A[p] and A[hi]
    store = lo
    for t in range(lo, hi):
        if A[t] < pivot:
            swap A[t] and A[store]
            store = store + 1
    swap A[store] and A[hi]
    return store    # final index of pivot
```

Step 4: Trace String Construction

- Given:
  ```
  intermediates = [(a1, b1), (a2, b2), ..., (am, bm)]
  final_state  = (af, bf)
  ```
- Produce:
  ```
  if m > 0:
      "(a1, b1), (a2, b2), ..., (am, bm) | (af, bf)"
  else:
      "| (af, bf)"
  ```

A.7 Divide and Conquer

Input

- **algorithm:** `<algorithm_name>`
- **input:** `<input_data>`
- **goal:** brief one-sentence description of what the algorithm should compute

Step 1: Identify Pattern

- **Pattern:** `binary_split`, `cross_subproblem`, `multi_way`, or `recursive_reduction`.
- **Why:** one sentence explaining why this divide-and-conquer pattern applies to the current problem.

Step 2: Base Case

```
If size <= <threshold>:
    return <direct computation>
```

Step 3: Divide

binary_split / cross_subproblem
```
mid = len(input) // 2
left_part  = input[:mid]
right_part = input[mid:]
```

multi_way
```
pivot = <pivot selection>
parts = partition(input, pivot)  # e.g., [< pivot, = pivot, > pivot]
```

recursive_reduction
```
next_input = <reduced input based on condition>
```

Applied to the current input, explicitly show the division:

```
Applied to input:
    <show actual division with concrete values>
```

Step 4: Conquer

binary_split / cross_subproblem
```
left_solution  = solve(left_part)
right_solution = solve(right_part)
```

multi_way
```
solutions = [solve(part) for part in parts]
```

recursive_reduction
```
solution = solve(next_input)
```

Record a concise trace of subproblems and results:

```
Trace:
    solve(<subproblem_1>) -> <result_1>
    solve(<subproblem_2>) -> <result_2>
    ...
```

Step 5: Combine

binary_split
```
final_result = merge(left_solution, right_solution)
```

cross_subproblem `cross_solution = compute_cross(input, mid)`

```
# show key computation with intermediate values
```

Example candidate comparison table:

Candidate	Value
left	
right	
cross	

```
final_result = best(left_solution, right_solution, cross_solution)
```

multi_way `final_result = combine(solutions)  # or select relevant partition`

recursive_reduction `final_result = solution  # or apply final transformation if needed`

Explicitly state the final value:

```
final_result = <value>
```

Step 6: Verify

```
Check:
    <brief confirmation that final_result satisfies the goal>
```

Answer

```
<answer><result></answer>
```

A.8 Greedy

Input

- **algorithm:** <algorithm_name>
  e.g., "Activity Selection", "Task Scheduling"
- **goal:** <maximize/minimize> <quantity>
  e.g., "maximize non-overlapping activities", "minimize weighted completion time"
- **items:** list of objects, e.g. [{id: 0, ...}, {id: 1, ...}, ...]

Step 1: Problem Understanding

- **Objective:** what quantity the algorithm is trying to optimize (maximize or minimize).
- **Constraint:** what makes a selection valid (e.g., no overlapping intervals, precedence constraints).

Step 2: Greedy Strategy

- **Sort by:** <property> (ascending or descending)
  - Activity Selection: sort by finish time (ascending).
  - Task Scheduling: sort by processing time (ascending) or ratio weight/time (descending).
- **Why:** one sentence justification for why this sort order leads to an optimal greedy choice.
- **Feasibility rule:** a condition comparing an item property to the current state, e.g. item.<property> <condition> state.<variable>
- **Invariant:** a condition that remains true after each selection, ensuring validity of the partial solution.

Step 3: Initialize

```
selected = []
state = { <variable>: <initial_value> }
```

Step 4: Sort

```
items_sorted = [
  { id: _, ... },  # rank 0
  { id: _, ... },  # rank 1
  ...
]
```

Step 5: Iterate

Use a step-by-step table to record the greedy decision process for each ranked item:

Rank	ID	Check	Result	Action	State Update
0					
1					
2					
⋮					

For each row:

- **Rank:** position in items_sorted.
- **ID:** item identifier.
- **Check:** feasibility condition evaluated for this item (e.g., start_time >= state.last_finish).
- **Result:** whether the condition is satisfied (True/False).

- **Action:** `"select"` or `"skip"`.
- **State Update:** how `state` changes if the item is selected.

Step 6: Verify

- **Selected:** list of chosen item IDs, e.g. `selected = [. . . ]`.
- **Constraint check:** brief confirmation that all selected items are jointly valid under the problem constraints (e.g., no overlaps, total capacity respected).

Answer

```
selected = [<ids>]
value    = <total>

<answer>[n values: position i = 1 if item i selected, else 0]</answer>
```