IMPERIAL

NLP Reading Group

December 3rd 2024

“Test Time Computation”

Test Time Computation Quick overview

Question: Given a fixed model & given a fixed test time example

How can we improve performance?

Scale Computation (**without** external feedback)

- Tree-of-thought
- ADaPT

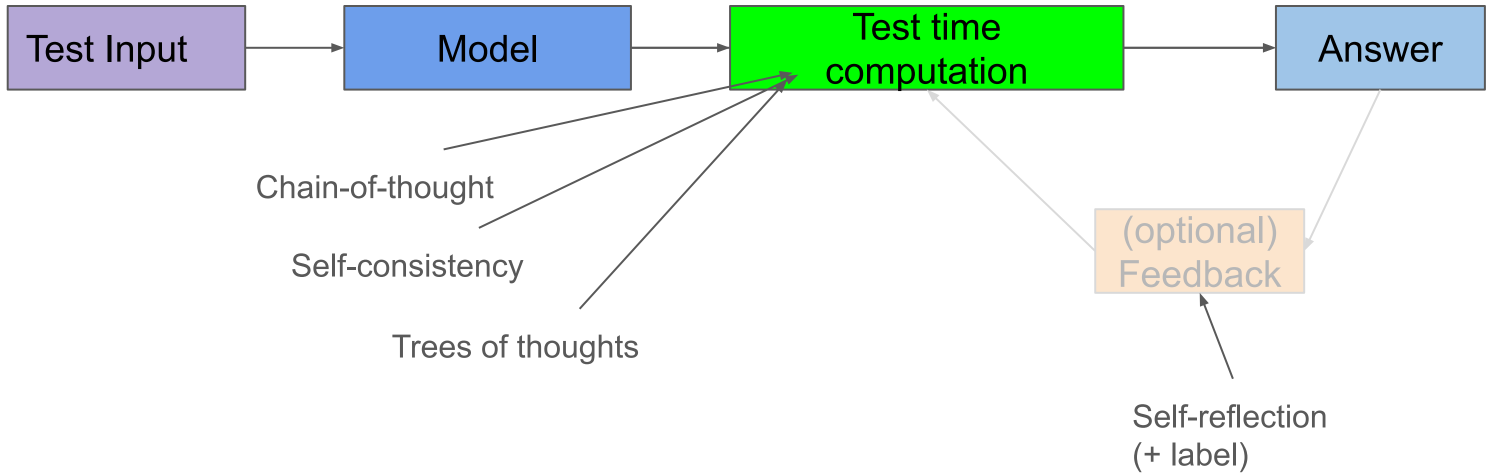
Scale Computation (**with** external feedback)

- Reflexion
- LATS
- (AdaPlanner)

Today

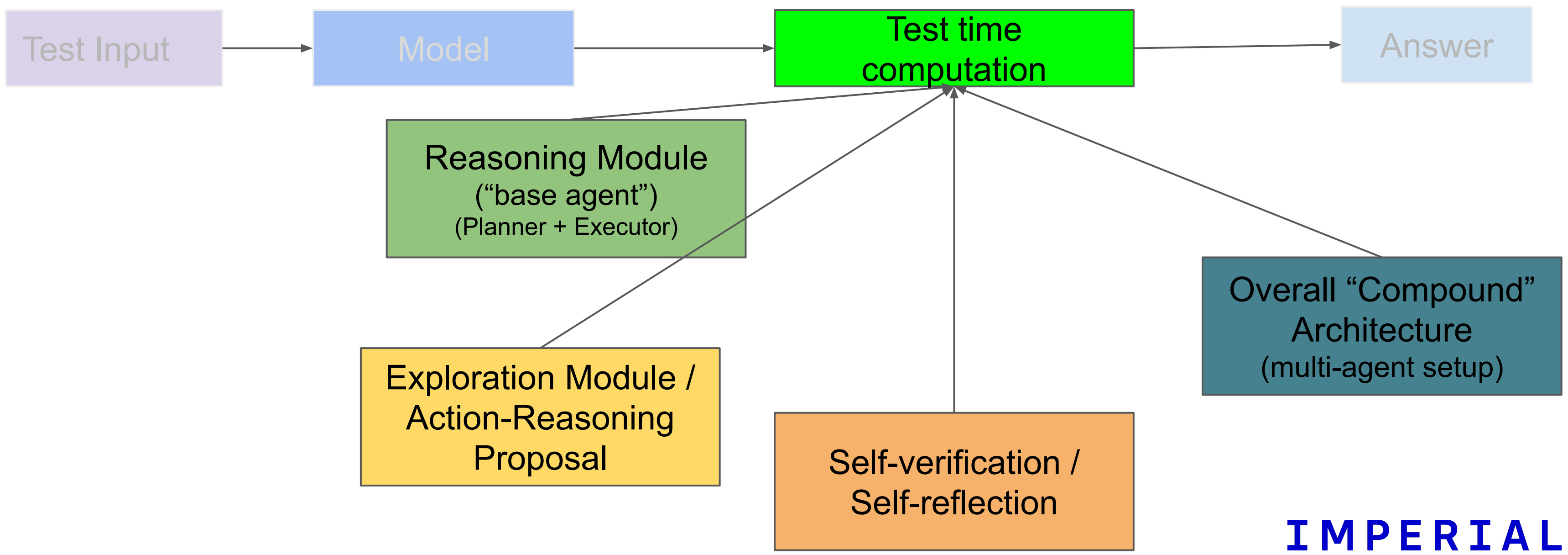
IMPERIAL

Test Time Computation Quick overview



IMPERIAL

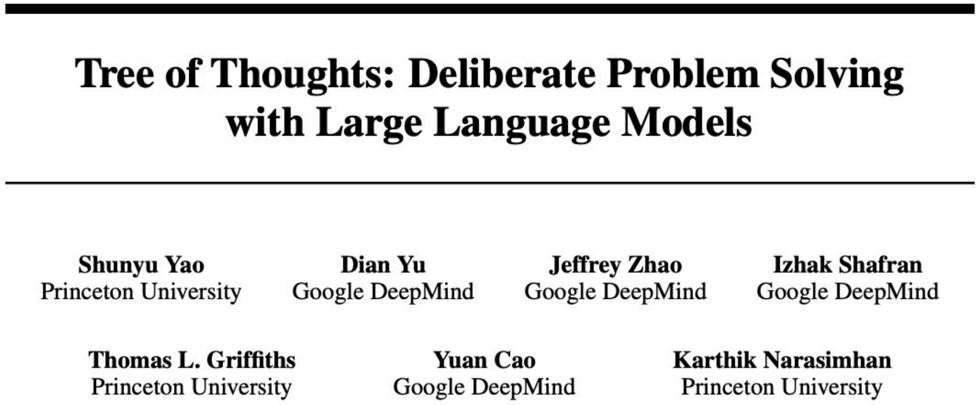
Test Time Computation Quick overview


Test Input
Model
Test time computation
Answer
Reasoning Module ("base agent") (Planner + Executor)
Exploration Module / Action-Reasoning Proposal
Self-verification / Self-reflection
Overall "Compound" Architecture (multi-agent setup)

IMPERIAL

Tree of Thoughts: Deliberate Problem Solving with Large Language Models

Tree of Thoughts: Deliberate Problem Solving with Large Language Models



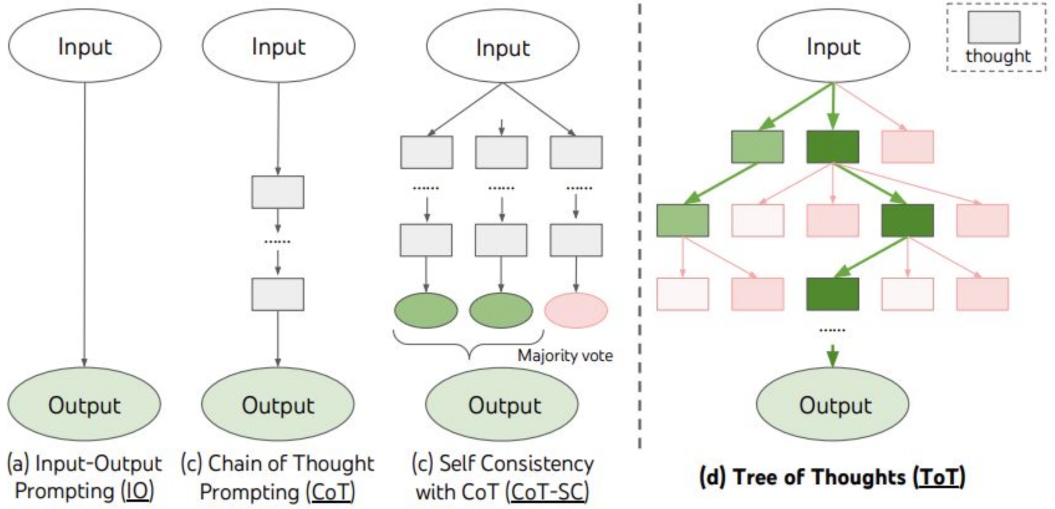
Tree of Thoughts: Deliberate Problem Solving with Large Language Models

Shunyu Yao, Princeton University
Dian Yu, Google DeepMind
Jeffrey Zhao, Google DeepMind
Izhak Shafran, Google DeepMind
Thomas L. Griffiths, Princeton University
Yuan Cao, Google DeepMind
Karthik Narasimhan, Princeton University

3rd December 2023

IMPERIAL

Tree of Thoughts: Deliberate Problem Solving with Large Language Models


Input
Output
(a) Input-Output Prompting (IO)
Input
Output
(c) Chain of Thought Prompting (CoT)
Input
Majority vote
Output
(c) Self Consistency with CoT (CoT-SC)
Input
thought
Output
(d) Tree of Thoughts (ToT)

IMPERIAL

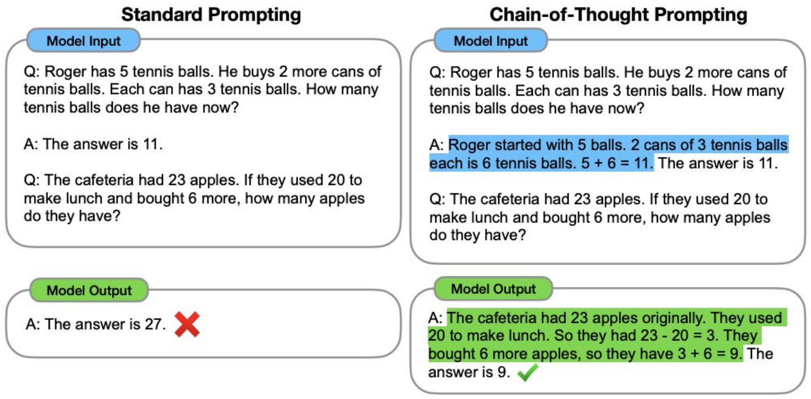
Tree of Thoughts: Deliberate Problem Solving with Large Language Models

Three main components:

1. Reasoning Steps (“thoughts”)
2. Reasoning proposers
3. Reasoning evaluators

Tree of Thoughts: Deliberate Problem Solving with Large Language Models

1. **Reasoning**



Standard Prompting

Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

Model Output

A: The answer is 27.

Chain-of-Thought Prompting

Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

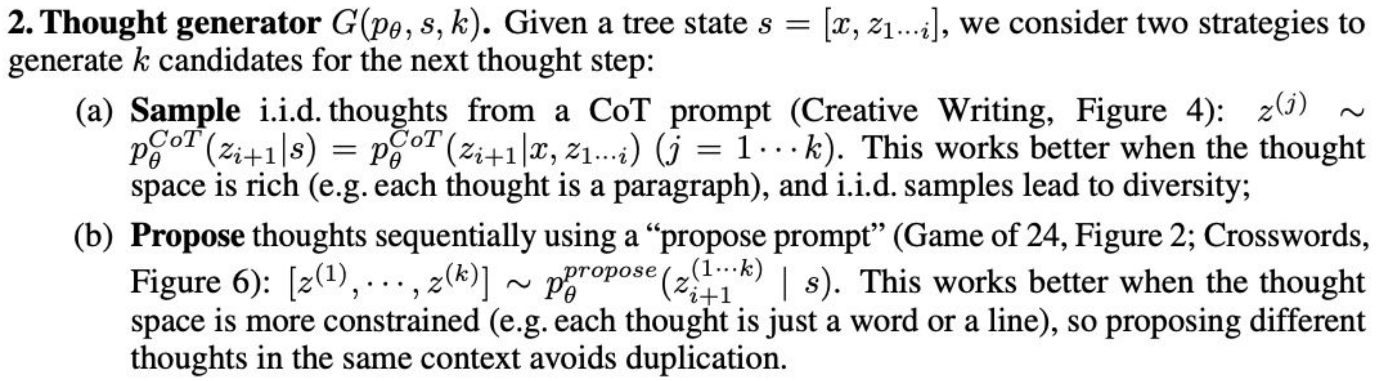
Model Output

A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9. The answer is 9.

“Classical Chain-of-thought”

Tree of Thoughts: Deliberate Problem Solving with Large Language Models

2. Reasoning Proposer



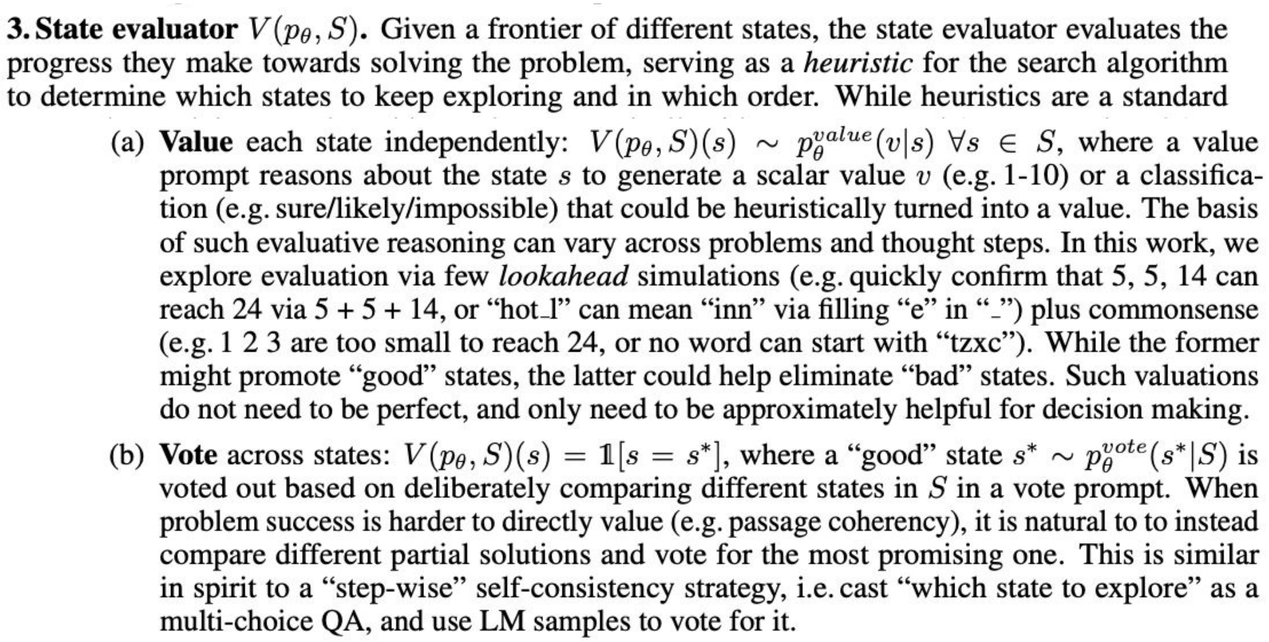
2. Thought generator $G(p_\theta, s, k)$**.** Given a tree state $s = [x, z_{1\cdots i}]$, we consider two strategies to generate k candidates for the next thought step:

(a) **Sample** i.i.d. thoughts from a CoT prompt (Creative Writing, Figure 4): $z^{(j)} \sim p_\theta^{CoT}(z_{i+1}|s) = p_\theta^{CoT}(z_{i+1}|x, z_{1\cdots i})$ $(j = 1 \cdots k)$. This works better when the thought space is rich (e.g. each thought is a paragraph), and i.i.d. samples lead to diversity;

(b) **Propose** thoughts sequentially using a "propose prompt" (Game of 24, Figure 2; Crosswords, Figure 6): $[z^{(1)}, \cdots, z^{(k)}] \sim p_\theta^{propose}(z_{i+1}^{(1\cdots k)} \mid s)$. This works better when the thought space is more constrained (e.g. each thought is just a word or a line), so proposing different thoughts in the same context avoids duplication.

Tree of Thoughts: Deliberate Problem Solving with Large Language Models

3. Reasoning Evaluator



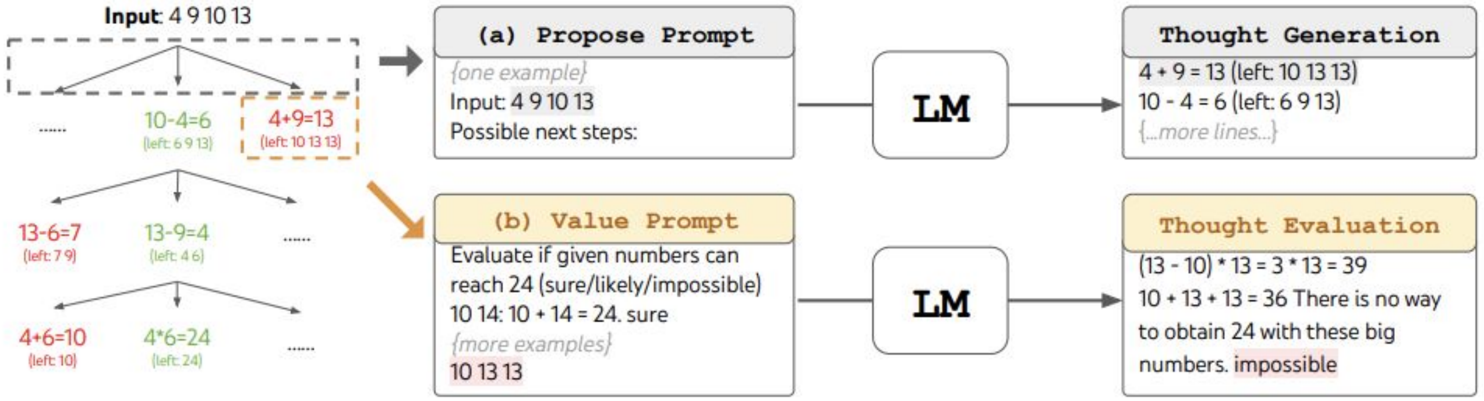
3. State evaluator $V(p_\theta, S)$**.** Given a frontier of different states, the state evaluator evaluates the progress they make towards solving the problem, serving as a *heuristic* for the search algorithm to determine which states to keep exploring and in which order. While heuristics are a standard

(a) **Value** each state independently: $V(p_\theta, S)(s) \sim p_\theta^{value}(v|s)\ \forall s \in S$, where a value prompt reasons about the state s to generate a scalar value v (e.g. 1-10) or a classification (e.g. sure/likely/impossible) that could be heuristically turned into a value. The basis of such evaluative reasoning can vary across problems and thought steps. In this work, we explore evaluation via few *lookahead* simulations (e.g. quickly confirm that 5, 5, 14 can reach 24 via 5 + 5 + 14, or "hot_l" can mean "inn" via filling "e" in "_") plus commonsense (e.g. 1 2 3 are too small to reach 24, or no word can start with "tzxc"). While the former might promote "good" states, the latter could help eliminate "bad" states. Such valuations do not need to be perfect, and only need to be approximately helpful for decision making.

(b) **Vote** across states: $V(p_\theta, S)(s) = \mathbb{1}[s = s^*]$, where a "good" state $s^* \sim p_\theta^{vote}(s^*|S)$ is voted out based on deliberately comparing different states in S in a vote prompt. When problem success is harder to directly value (e.g. passage coherency), it is natural to to instead compare different partial solutions and vote for the most promising one. This is similar in spirit to a "step-wise" self-consistency strategy, i.e. cast "which state to explore" as a multi-choice QA, and use LM samples to vote for it.

Tree of Thoughts: Deliberate Problem Solving with Large Language Models

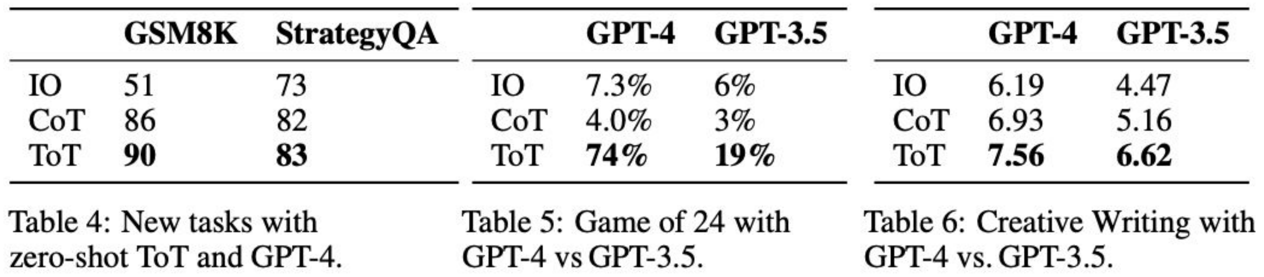
Example:



IMPERIAL

Tree of Thoughts: Deliberate Problem Solving with Large Language Models

Results



	GSM8K	StrategyQA
IO	51	73
CoT	86	82
ToT	**90**	**83**

Table 4: New tasks with zero-shot ToT and GPT-4.

	GPT-4	GPT-3.5
IO	7.3%	6%
CoT	4.0%	3%
ToT	**74%**	**19%**

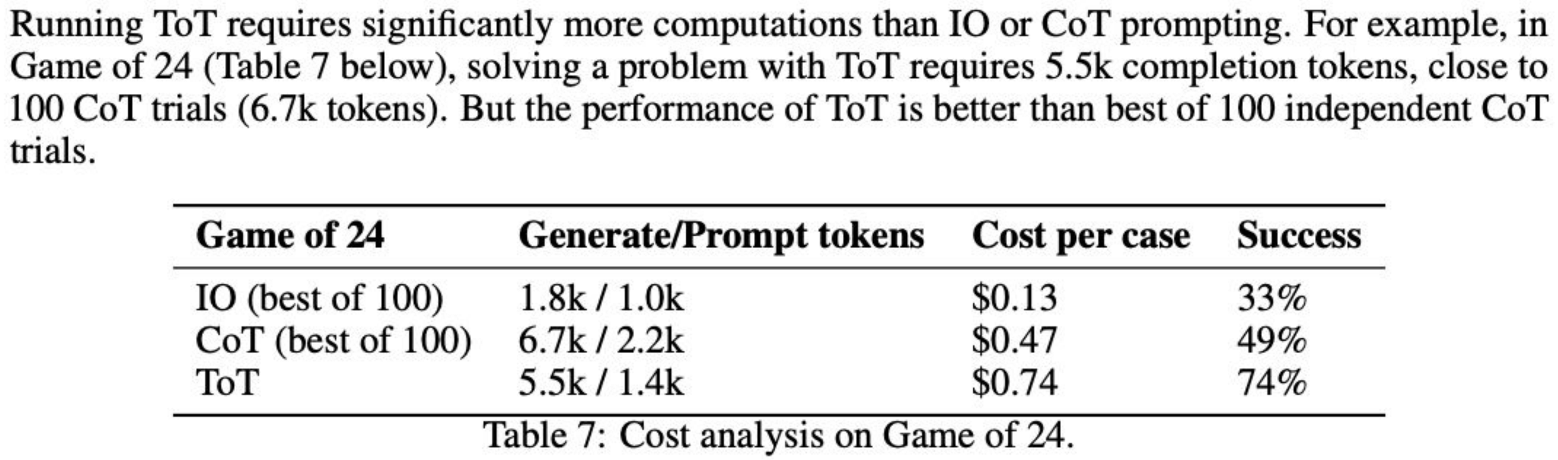
Table 5: Game of 24 with GPT-4 vs GPT-3.5.

	GPT-4	GPT-3.5
IO	6.19	4.47
CoT	6.93	5.16
ToT	**7.56**	**6.62**

Table 6: Creative Writing with GPT-4 vs. GPT-3.5.

Tree of Thoughts: Deliberate Problem Solving with Large Language Models

Cost & Efficiency



Running ToT requires significantly more computations than IO or CoT prompting. For example, in Game of 24 (Table 7 below), solving a problem with ToT requires 5.5k completion tokens, close to 100 CoT trials (6.7k tokens). But the performance of ToT is better than best of 100 independent CoT trials.

Game of 24	Generate/Prompt tokens	Cost per case	Success
IO (best of 100)	1.8k / 1.0k	\$0.13	33%
CoT (best of 100)	6.7k / 2.2k	\$0.47	49%
ToT	5.5k / 1.4k	\$0.74	74%

Table 7: Cost analysis on Game of 24.

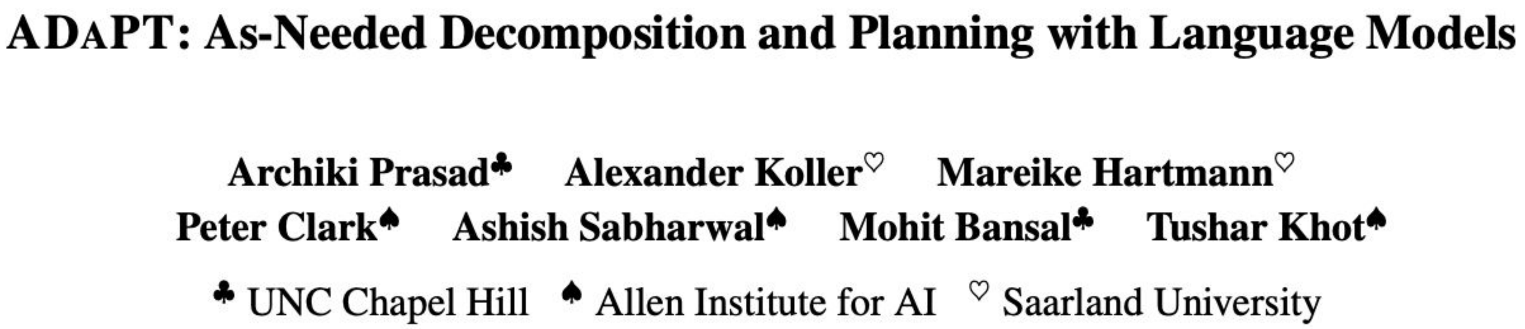
Tree of Thoughts: Deliberate Problem Solving with Large Language Models

Conclusion:

1. **Effective method of scaling test time computation to improve score**
2. **Some tasks are affected less (GSM** [saturation] **& Creative Writing** [not a ‘logical’ task?]**)**
3. **Would be interesting to have proper cost analysis of computation**

ADaPT: As-Needed Decomposition and Planning with Language Models

ADaPT: As-Needed Decomposition and Planning with Language Models



ADaPT: As-Needed Decomposition and Planning with Language Models

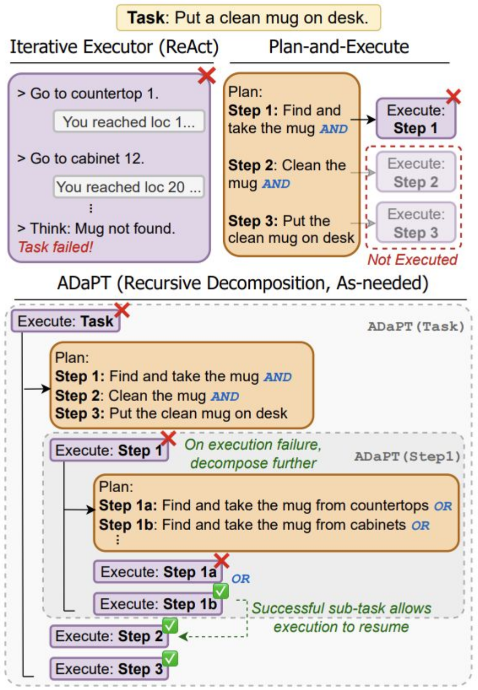
Archiki Prasad[♣] Alexander Koller[♡] Mareike Hartmann[♡]
Peter Clark[♠] Ashish Sabharwal[♠] Mohit Bansal[♣] Tushar Khot[♠]

[♣] UNC Chapel Hill [♠] Allen Institute for AI [♡] Saarland University

8th April 2024

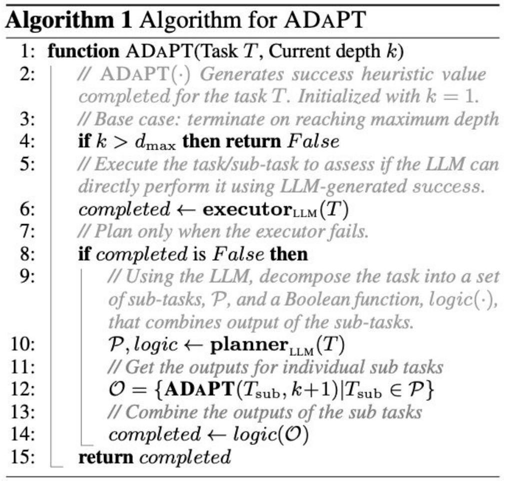
IMPERIAL

ADaPT: As-Needed Decomposition and Planning with Language Models



Task: Put a clean mug on desk.
Iterative Executor (ReAct)
Plan-and-Execute
> Go to countertop 1.
You reached loc 1...
> Go to cabinet 12.
You reached loc 20 ...
> Think: Mug not found.
Task failed!
Plan:
Step 1: Find and take the mug AND
Step 2: Clean the mug AND
Step 3: Put the clean mug on desk
Execute: Step 1
Execute: Step 2
Execute: Step 3
Not Executed
ADaPT (Recursive Decomposition, As-needed)
Execute: Task
ADaPT(Task)
Plan:
Step 1: Find and take the mug AND
Step 2: Clean the mug AND
Step 3: Put the clean mug on desk
Execute: Step 1
On execution failure, decompose further
ADaPT(Step1)
Plan:
Step 1a: Find and take the mug from countertops OR
Step 1b: Find and take the mug from cabinets OR
Execute: Step 1a
OR
Execute: Step 1b
Successful sub-task allows execution to resume
Execute: Step 2
Execute: Step 3

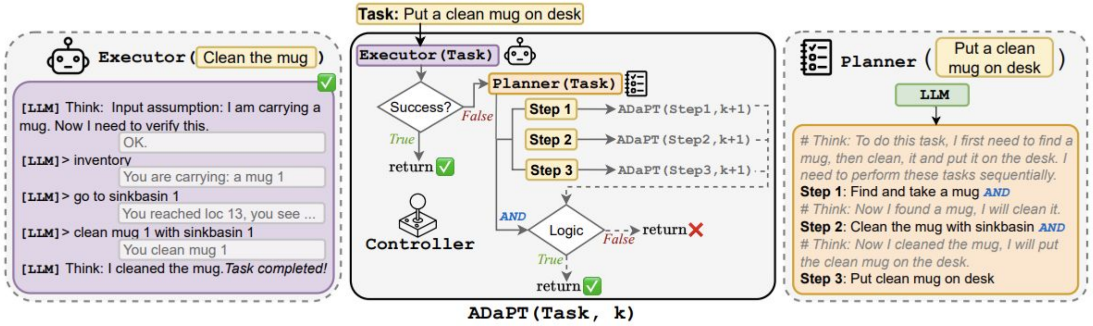
ADaPT: As-Needed Decomposition and Planning with Language Models



Algorithm 1 Algorithm for ADaPT

```
1:  function ADaPT(Task $T$, Current depth $k$)
2:      // ADaPT($\cdot$) Generates success heuristic value completed for the task $T$. Initialized with $k = 1$.
3:      // Base case: terminate on reaching maximum depth
4:      if $k > d_{\max}$ then return $False$
5:      // Execute the task/sub-task to assess if the LLM can directly perform it using LLM-generated success.
6:      $completed \leftarrow \textbf{executor}_{\text{LLM}}(T)$
7:      // Plan only when the executor fails.
8:      if $completed$ is $False$ then
9:          // Using the LLM, decompose the task into a set of sub-tasks, $\mathcal{P}$, and a Boolean function, $logic(\cdot)$, that combines output of the sub-tasks.
10:         $\mathcal{P}, logic \leftarrow \textbf{planner}_{\text{LLM}}(T)$
11:         // Get the outputs for individual sub tasks
12:         $\mathcal{O} = \{\textbf{ADaPT}(T_{\text{sub}}, k+1) | T_{\text{sub}} \in \mathcal{P}\}$
13:         // Combine the outputs of the sub tasks
14:         $completed \leftarrow logic(\mathcal{O})$
15:     return $completed$
```

Key Components:
1. LLM Planner
2. LLM Executor
3. LLM Verifier
4. Overall Controller



[LLM] Think: Input assumption: I am carrying a mug. Now I need to verify this.
OK.
[LLM] > inventory
You are carrying: a mug 1
[LLM] > go to sinkbasin 1
You reached loc 13, you see ...
[LLM] > clean mug 1 with sinkbasin 1
You clean mug 1
[LLM] Think: I cleaned the mug. *Task completed!*

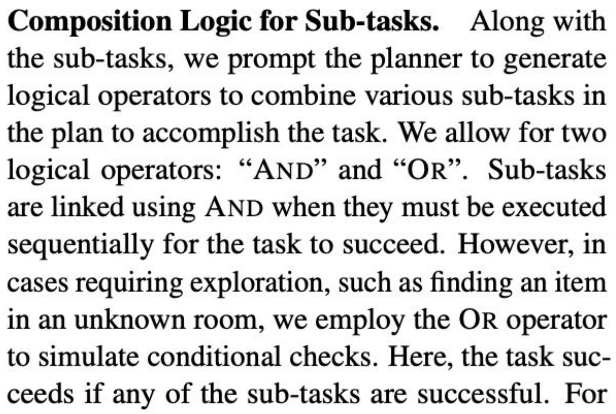
# Think: To do this task, I first need to find a mug, then clean, it and put it on the desk. I need to perform these tasks sequentially.
Step 1: Find and take a mug ***AND***
# Think: Now I found a mug, I will clean it.
Step 2: Clean the mug with sinkbasin ***AND***
# Think: Now I cleaned the mug, I will put the clean mug on the desk.
Step 3: Put clean mug on desk

Figure 2: Block diagram of the ADaPT pipeline with an example from ALFWorld. **Left:** Use of LLM as an executor to interact iteratively with the environment along with an example execution trajectory. **Middle:** Overall recursive algorithm (depth $k \leq d_{\max}$) that embeds the executor and planner, refer to Algorithm 1 for details. **Right:** Outline of using LLM as a planner to generate sub-tasks (steps) and logical operators combining them.

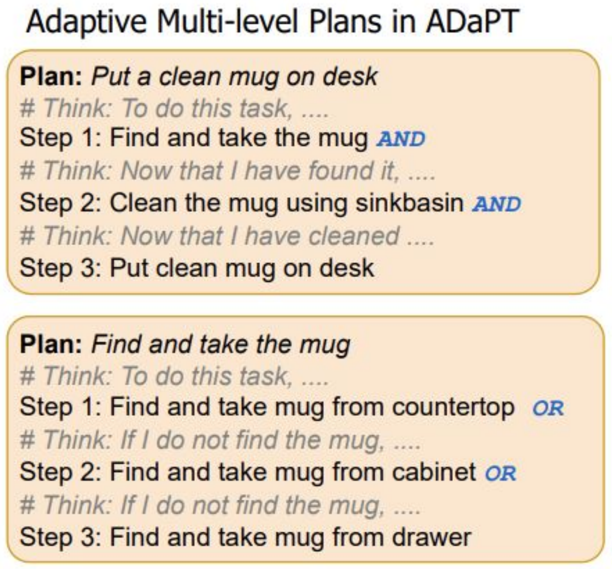
IMPERIAL

ADaPT: As-Needed Decomposition and Planning with Language Models

1. Planner



Composition Logic for Sub-tasks. Along with the sub-tasks, we prompt the planner to generate logical operators to combine various sub-tasks in the plan to accomplish the task. We allow for two logical operators: "AND" and "OR". Sub-tasks are linked using AND when they must be executed sequentially for the task to succeed. However, in cases requiring exploration, such as finding an item in an unknown room, we employ the OR operator to simulate conditional checks. Here, the task succeeds if any of the sub-tasks are successful. For



Adaptive Multi-level Plans in ADaPT

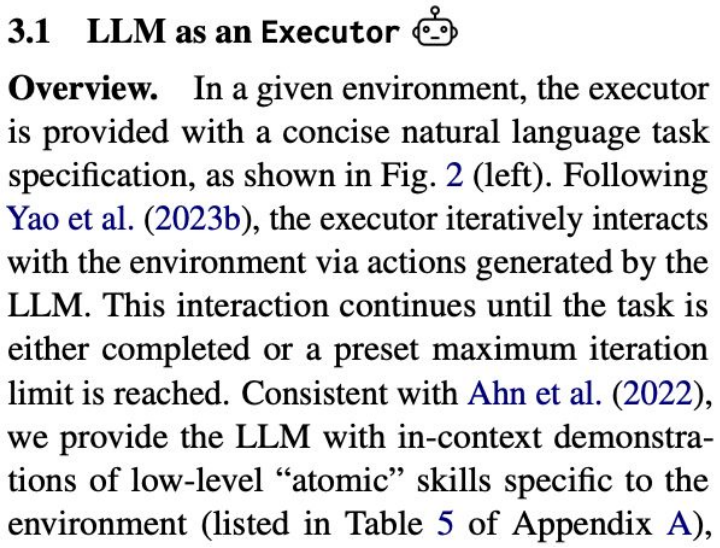
Plan: *Put a clean mug on desk*
# Think: To do this task, ...
Step 1: Find and take the mug ***AND***
# Think: Now that I have found it, ...
Step 2: Clean the mug using sinkbasin ***AND***
# Think: Now that I have cleaned ....
Step 3: Put clean mug on desk

Plan: *Find and take the mug*
# Think: To do this task, ...
Step 1: Find and take mug from countertop ***OR***
# Think: If I do not find the mug, ...
Step 2: Find and take mug from cabinet ***OR***
# Think: If I do not find the mug, ...
Step 3: Find and take mug from drawer

IMPERIAL

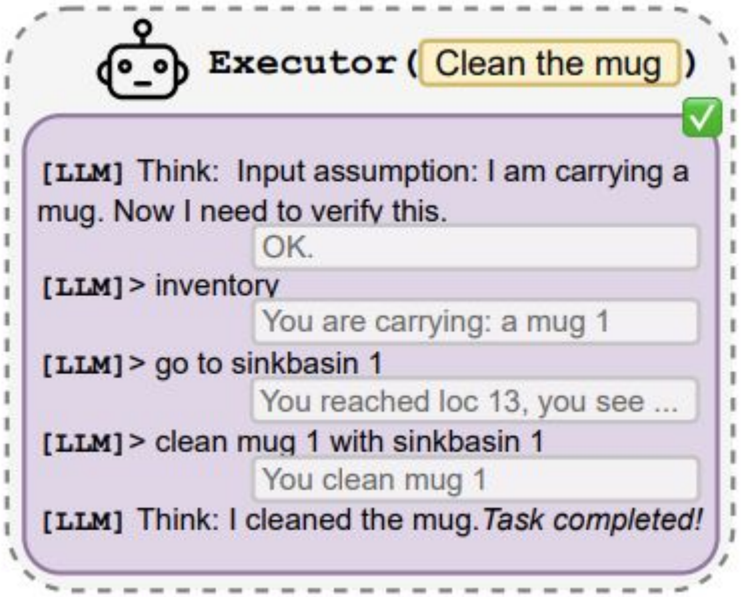
ADaPT: As-Needed Decomposition and Planning with Language Models

2. Executor



3.1 LLM as an Executor

Overview. In a given environment, the executor is provided with a concise natural language task specification, as shown in Fig. 2 (left). Following Yao et al. (2023b), the executor iteratively interacts with the environment via actions generated by the LLM. This interaction continues until the task is either completed or a preset maximum iteration limit is reached. Consistent with Ahn et al. (2022), we provide the LLM with in-context demonstrations of low-level "atomic" skills specific to the environment (listed in Table 5 of Appendix A),

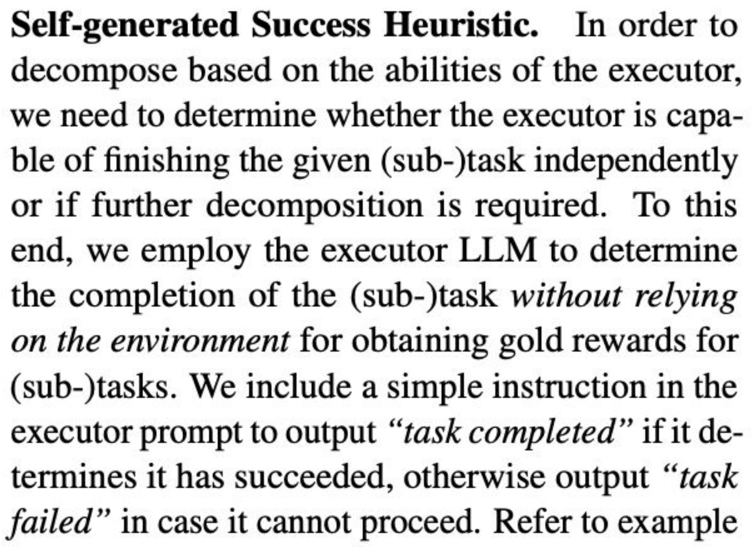


```
[LLM] Think: Input assumption: I am carrying a mug. Now I need to verify this.
                                   OK.
[LLM] > inventory
                                   You are carrying: a mug 1
[LLM] > go to sinkbasin 1
                                   You reached loc 13, you see ...
[LLM] > clean mug 1 with sinkbasin 1
                                   You clean mug 1
[LLM] Think: I cleaned the mug. Task completed!
```

IMPERIAL

ADaPT: As-Needed Decomposition and Planning with Language Models

3. Verifier



Self-generated Success Heuristic. In order to decompose based on the abilities of the executor, we need to determine whether the executor is capable of finishing the given (sub-)task independently or if further decomposition is required. To this end, we employ the executor LLM to determine the completion of the (sub-)task *without relying on the environment* for obtaining gold rewards for (sub-)tasks. We include a simple instruction in the executor prompt to output *"task completed"* if it determines it has succeeded, otherwise output *"task failed"* in case it cannot proceed. Refer to example

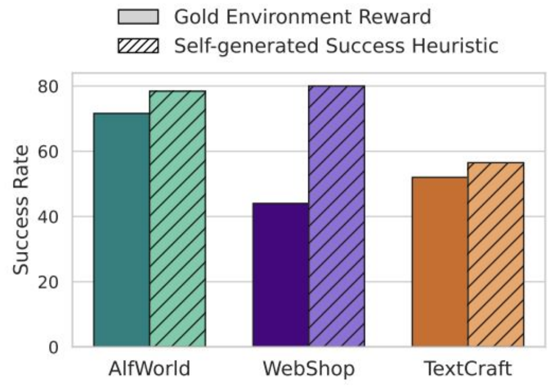
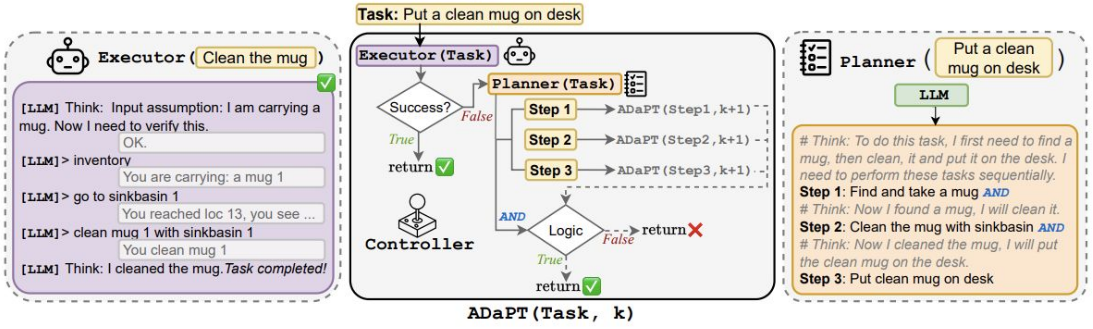


Figure 12: Comparison of LLM-generated success heuristic with gold environment rewards to compute success rates for all datasets.

ADaPT: As-Needed Decomposition and Planning with Language Models

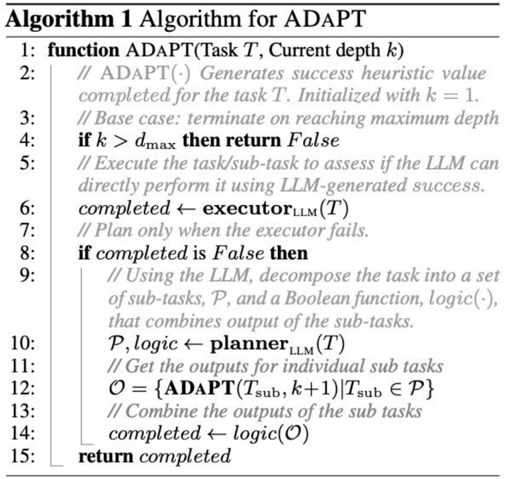
4. Overall programme



[LLM] Think: Input assumption: I am carrying a mug. Now I need to verify this.
OK.
[LLM] > inventory
You are carrying: a mug 1
[LLM] > go to sinkbasin 1
You reached loc 13, you see ...
[LLM] > clean mug 1 with sinkbasin 1
You clean mug 1
[LLM] Think: I cleaned the mug. *Task completed!*

# Think: To do this task, I first need to find a mug, then clean, it and put it on the desk. I need to perform these tasks sequentially.
Step 1: Find and take a mug *AND*
# Think: Now I found a mug, I will clean it.
Step 2: Clean the mug with sinkbasin *AND*
# Think: Now I cleaned the mug, I will put the clean mug on the desk.
Step 3: Put clean mug on desk

Figure 2: Block diagram of the ADAPT pipeline with an example from ALFWorld. **Left:** Use of LLM as an executor to interact iteratively with the environment along with an example execution trajectory. **Middle:** Overall recursive algorithm (depth $k \leq d_{\max}$) that embeds the executor and planner, refer to Algorithm 1 for details. **Right:** Outline of using LLM as a planner to generate sub-tasks (steps) and logical operators combining them.



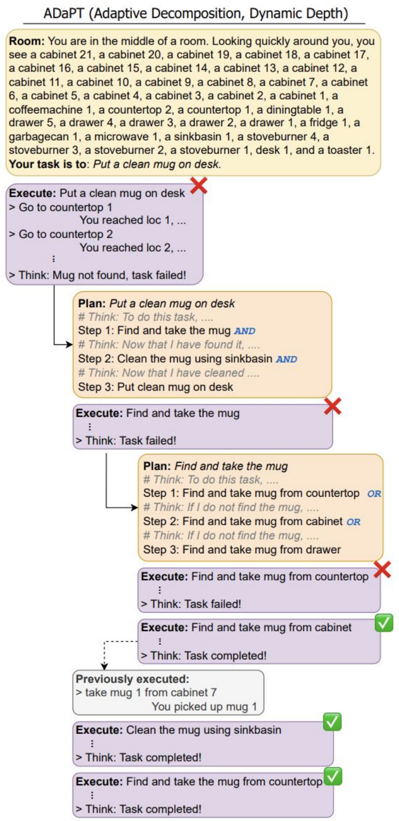
Algorithm 1 Algorithm for ADAPT

```
1:  function ADAPT(Task $T$, Current depth $k$)
2:      // ADAPT($\cdot$) Generates success heuristic value completed for the task $T$. Initialized with $k = 1$.
3:      // Base case: terminate on reaching maximum depth
4:      if $k > d_{\max}$ then return $False$
5:      // Execute the task/sub-task to assess if the LLM can directly perform it using LLM-generated success.
6:      $completed \leftarrow \mathbf{executor}_{\text{LLM}}(T)$
7:      // Plan only when the executor fails.
8:      if $completed$ is $False$ then
9:          // Using the LLM, decompose the task into a set of sub-tasks, $\mathcal{P}$, and a Boolean function, $logic(\cdot)$, that combines output of the sub-tasks.
10:         $\mathcal{P}, logic \leftarrow \mathbf{planner}_{\text{LLM}}(T)$
11:         // Get the outputs for individual sub tasks
12:         $\mathcal{O} = \{\mathbf{ADAPT}(T_{\text{sub}}, k+1) | T_{\text{sub}} \in \mathcal{P}\}$
13:         // Combine the outputs of the sub tasks
14:         $completed \leftarrow logic(\mathcal{O})$
15:     return $completed$
```

IMPERIAL

ADaPT: As-Needed Decomposition and Planning with Language Models

Example:



ADaPT (Adaptive Decomposition, Dynamic Depth)

Room: You are in the middle of a room. Looking quickly around you, you see a cabinet 21, a cabinet 20, a cabinet 19, a cabinet 18, a cabinet 17, a cabinet 16, a cabinet 15, a cabinet 14, a cabinet 13, a cabinet 12, a cabinet 11, a cabinet 10, a cabinet 9, a cabinet 8, a cabinet 7, a cabinet 6, a cabinet 5, a cabinet 4, a cabinet 3, a cabinet 2, a cabinet 1, a coffeemachine 1, a countertop 2, a countertop 1, a diningtable 1, a drawer 5, a drawer 4, a drawer 3, a drawer 2, a drawer 1, a fridge 1, a garbagecan 1, a microwave 1, a sinkbasin 1, a stoveburner 4, a stoveburner 3, a stoveburner 2, a stoveburner 1, desk 1, and a toaster 1.
Your task is to: *Put a clean mug on desk.*

Execute: Put a clean mug on desk ✗
> Go to countertop 1
You reached loc 1, ...
> Go to countertop 2
You reached loc 2, ...
⋮
> Think: Mug not found, task failed!

Plan: *Put a clean mug on desk*
# Think: To do this task, ...
Step 1: Find and take the mug *AND*
# Think: Now that I have found it, ...
Step 2: Clean the mug using sinkbasin *AND*
# Think: Now that I have cleaned ....
Step 3: Put clean mug on desk

Execute: Find and take the mug ✗
⋮
> Think: Task failed!

Plan: *Find and take the mug*
# Think: To do this task, ...
Step 1: Find and take mug from countertop *OR*
# Think: If I do not find the mug, ...
Step 2: Find and take mug from cabinet *OR*
# Think: If I do not find the mug, ...
Step 3: Find and take mug from drawer

Execute: Find and take mug from countertop ✗
⋮
> Think: Task failed!

Execute: Find and take mug from cabinet ✅
⋮
> Think: Task completed!

Previously executed:
> take mug 1 from cabinet 7
You picked up mug 1

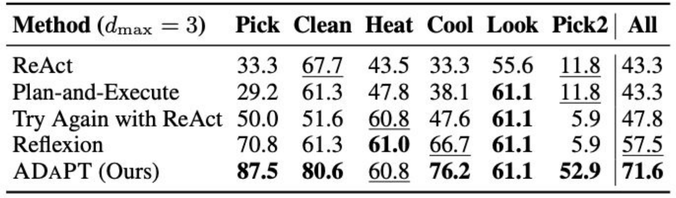
Execute: Clean the mug using sinkbasin ✅
⋮
> Think: Task completed!

Execute: Find and take the mug from countertop ✅
⋮
> Think: Task completed!

IMPERIAL

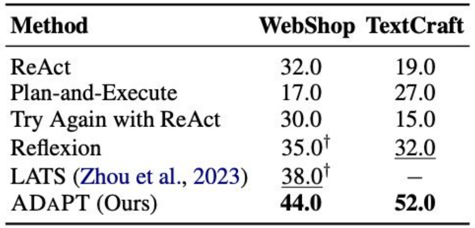
ADaPT: As-Needed Decomposition and Planning with Language Models

Results:



Method ($d_{\max} = 3$)	Pick	Clean	Heat	Cool	Look	Pick2	All
ReAct	33.3	67.7	43.5	33.3	55.6	11.8	43.3
Plan-and-Execute	29.2	61.3	47.8	38.1	**61.1**	11.8	43.3
Try Again with ReAct	50.0	51.6	60.8	47.6	**61.1**	5.9	47.8
Reflexion	70.8	61.3	**61.0**	66.7	**61.1**	5.9	57.5
ADaPT (Ours)	**87.5**	**80.6**	60.8	**76.2**	**61.1**	**52.9**	**71.6**

Table 1: ADaPT yields the highest the overall success rates (%) compared to baselines from prior work (discussed in Sec. 4.2) on ALFWorld (test split). Best (highest) success rates are highlighted in bold and second-highest rates are underlined.



Method	WebShop	TextCraft
ReAct	32.0	19.0
Plan-and-Execute	17.0	27.0
Try Again with ReAct	30.0	15.0
Reflexion	35.0[†]	32.0
LATS (Zhou et al., 2023)	38.0[†]	–
ADaPT (Ours)	**44.0**	**52.0**

Table 2: ADaPT yields the highest success rate on WebShop and TextCraft (test split) with $d_{\max} = 3$ and 4 respectively. [†]Performance reported by Zhou et al. (2023)

IMPERIAL

ADaPT: As-Needed Decomposition and Planning with Language Models

Ablations:

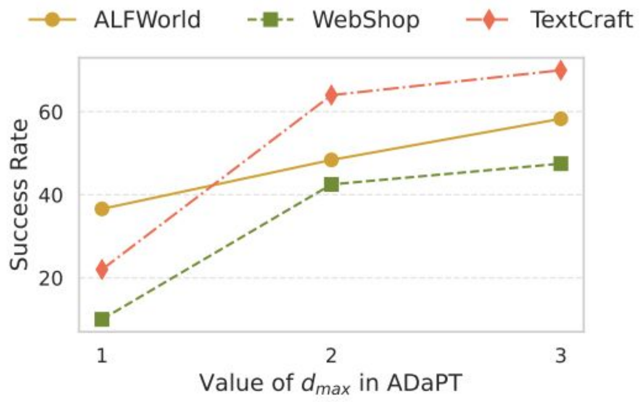


Figure 4: Success rate of ADaPT increases with the maximum depth $d_{\max}$ for all datasets (dev splits).

ADaPT: As-Needed Decomposition and Planning with Language Models

Cost:

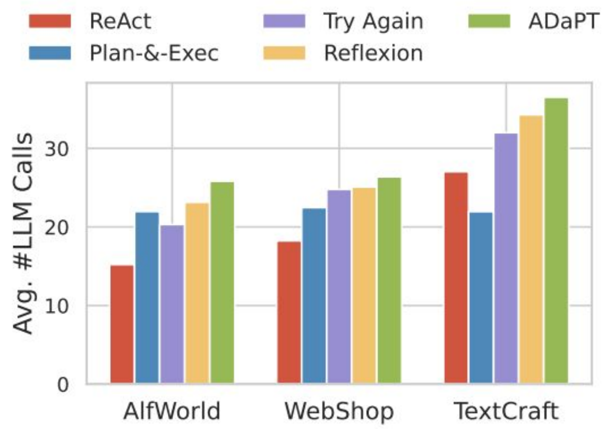


Figure 7: Average number of LLM calls for each approach including ADAPT and baselines discussed in Sec. 4.2 with GPT-3.5 LLM across datasets.

IMPERIAL

ADaPT: As-Needed Decomposition and Planning with Language Models

Conclusion:

1. **Effective method of scaling test time computation to improve score** (and works with agents)
2. **Seems to be cleverer than ToT (as only decomposes when needed)**
3. **Would be interesting to have proper cost analysis of computation**

Takeaways & Other ideas

IMPERIAL

Takeaways & Other Ideas

1. Test time computation can be very effective

2. Different ways to influence test-time computation (better base “reasoning”; better exploration; better “verification & reflection”; better overall architecture) (**collaboration anyone?**)

3. It would be interesting to have a good analysis of cost vs. performance across methods (**collaboration anyone?**)

4. Anything else?

IMPERIAL

Bibliography

- CoT: https://arxiv.org/pdf/2201.11903
- ToT: https://arxiv.org/pdf/2305.10601
- LATS: https://arxiv.org/pdf/2310.04406
- ADaPT: https://arxiv.org/pdf/2311.05772
- Reflexion: https://arxiv.org/abs/2303.11366
- AdaPlanner: https://arxiv.org/abs/2305.16653
- StateAct: https://arxiv.org/abs/2410.02810