Scaling laws: Test time compute

At first, it was thought that the way to achieve a high Performance Metric was via using models with a large number of parameters.

The Scaling Law showed this to not be true

- defines the Performance as a function of number of parameters and number of tokens consumed during training
- thus, a small and large model might achieve the same Performance
    - with the smaller model needing to consume many more tokens during training

But there is another dimension to consider in order to reach a Performance target

- amount of compute used at *test/inference* time.

Increasing Performance by increasing compute at inference time

There are several techniques for improving Performance by using more resources at inference time.

- *Test-time scaling*
- *Reasoning*: Training a model to "think" longer

Test-time scaling refers to

- increasing resources purely at inference time

This is often manifested

- using a pre-trained LLM as a "subroutine" in a control program
- enabling the LLM to produce multiple candidate answers
- which are filtered into a final answer

Reasoning

- uses *training-time* resources
- to train an LLM to produce longer answers
    - which also results in more test-time compute being consumed

These longer answers are not merely increasing verbosity

- they encourage the LLM to generate a plan for creating an answer
- which culminates in the final answer

The plan increases the probability that the final answer is correct (increase in the Performance metric).

These two methods may be combined

- training a model to Reason
    - using training-time compute
- using test-time strategies to augment the reasoning
    - searching for the best answer among multiple candidate, where each candidate answer uses reasoning
    - guiding the reasoning process
        - *back-tracking* when a reasoning trace appears unlikely to succeed
        - encouraging *revision* and *reflection*

We will devote a separate module to Reasoning.

This module will focus on pure test-time increases in compute.

Test-time scaling

We first focus on pure test-time increases in compute

- with no pre-training that uses additional training-time compute

An LLM can produce more than one solution

When predicting the next token (using a Transformer in auto-regressive mode) the Language Model outputs

$$\backslash\mathbf{pr}\backslash\mathbf{y}_{\backslash\mathrm{tp}} \mid \backslash\mathbf{y}_{(1:-1)}$$

- a *probability distribution*
- from which we sample a single token $\backslash\mathbf{y}_{\backslash\mathrm{tp}}$

If we sample deterministically

$$\backslash\hat{\mathrm{y}}_{\backslash\mathrm{tp}} = \backslash\mathrm{Vocab}_{j^*} \text{ where } j^* = \backslash\mathrm{argmax}j\backslash\mathrm{pr}\backslash\mathrm{y}_{\backslash\mathrm{tp}} \mid \backslash\mathrm{y}_{(1:-1)}{}_{j}$$

- i.e., the token in vocabulary $\backslash\mathrm{Vocab}$ with the greatest probability in probability vector $\backslash\mathrm{pr}\backslash\mathrm{y}_{\backslash\mathrm{tp}} \mid \backslash\mathrm{y}_{(1:-1)}$
- the next token is unique
- the answer sequence $\backslash\mathrm{y}$ is unique

But if we sample according to the probabilities $\backslash\mathbf{pr}\backslash\mathbf{y}_{\backslash\mathrm{tp}} \mid \backslash\mathbf{y}_{(1:-1)}$

- there are many possible answer sequences $\backslash\mathbf{y}$
- some of them syntactically different but with the same answer
- some with different answers

Thus, it is possible to use compute at test time to generate more than one answer to a prompt.

There are several strategies for using multiple answers to improve Performance

Multiple answer attempts can improve Performance

Suppose we allow the LLM multiple attempts at producing an answer.

Is is likely that one of the candidate answers is the correct one ?

Suppose we sample k answers and have a method for identifying the "best".

How does that affect Performance ?

Rather than grading the response to a prompt with a binary Pass/Fail grade

- Define *pass@k* to be true if one of the k candidate answers is correct.

We examine how *pass@k* varies with increasing k.

Coverage vs. number of samples

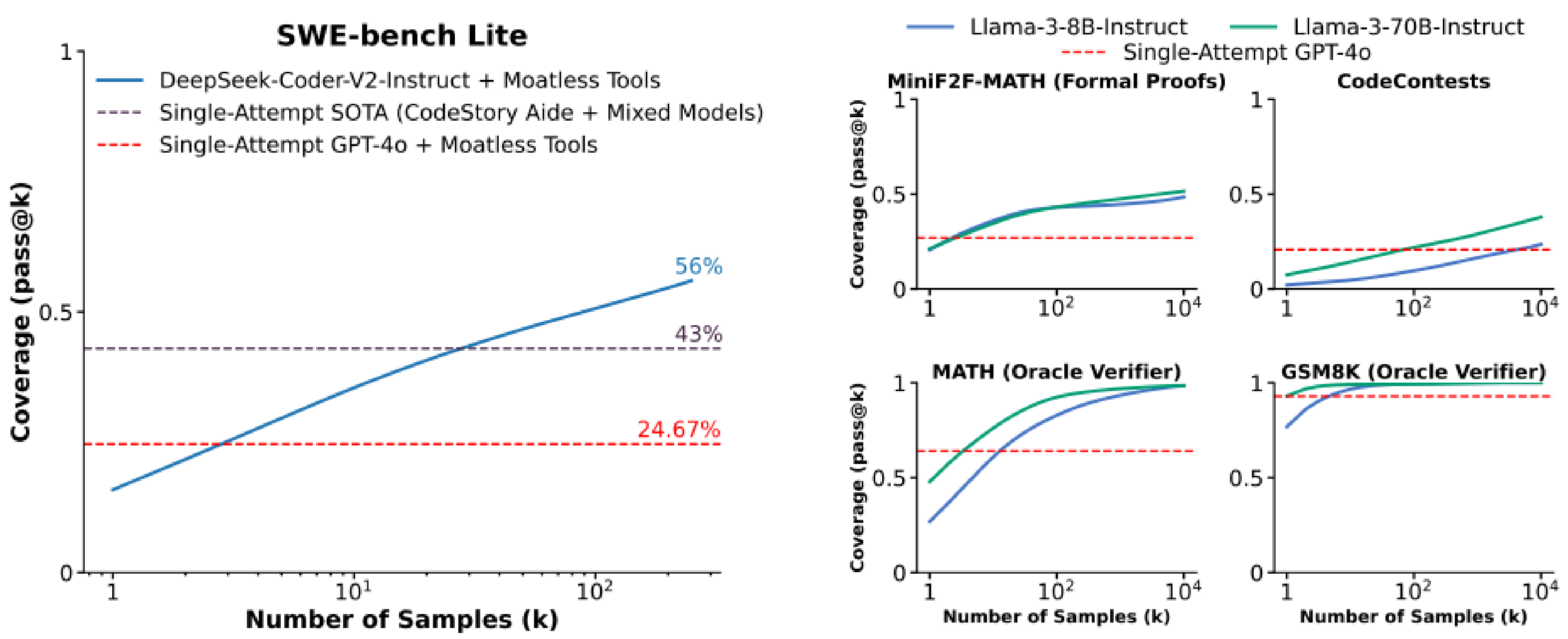


Figure 2: Across five tasks, we find that coverage (the fraction of problems solved by at least one generated sample) increases as we scale the number of samples. Notably, using repeated sampling, we are able to increase the solve rate of an open-source method from 15.9% to 56% on SWE-bench Lite.

Attribution: https://arxiv.org/pdf/2407.21787#page=5

The chart shows that

- if we can identify the correct answers from among the k candidates
- increasing Performance $P(k)$ by a constant increment α
- requires an exponential increase in k

That is

$$P(k) = \alpha \log_b k$$

so increasing k to $b * k$ boost Performance to $P(k) + \alpha$

$$\begin{aligned} P(k * b) &= \alpha \log_b k * b \\ &= \alpha * (\log_b k + 1) \\ &= P(k) + \alpha \end{aligned}$$

For example, referring to the above chart (right side, bottom row/left column)

- for MATH (Oracle Verifier) Llama-3-8B **blue line**
    - increasing Coverage from 0.25 to 0.5
    - requires increasing k from $10^0 = 1$ to 10^1

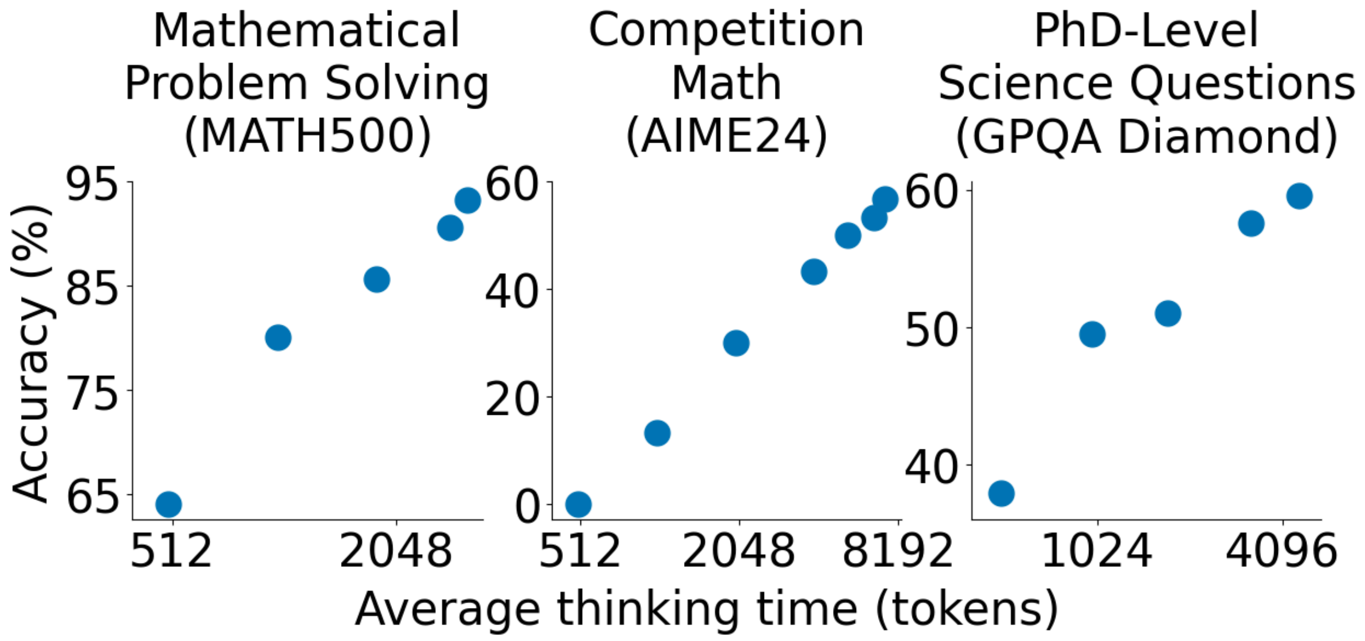
So test-time compute can be effective, but quite expensive.

Test-time Scaling Law

The simplest description of the Test-time Scaling law is the following chart

- Performance
- versus Compute Budget
  - log scale for compute "thinking time"

Test-Time Scaling Law
as function of length of reasoning trace



Attribution: https://arxiv.org/pdf/2501.19393#page=1 (Note the horizontal scale is logarithmic)

Note that the horizontal axis is $\log_2$ scale.

So the Scaling Law says that

- Performance is log-linear in Compute Budget
- so increasing Performance by a constant increment requires an exponential increase in compute

This is very expensive !

We will examine this in more detail below.

Scaling alternatives: Test-time versus Model Size

How does scaling via Test-Time compute

- compare to scaling by increasing the number of parameters ?
    - using training-time compute rather than test-time compute

The chart below compares the Performance of

- a small model using Test-Time Compute
- a model 14 times larger in parameters that only uses pre-training (no test-time compute)

when both models use the *same compute budget* (FLOPS-matched)

- the small model uses the budget at test-time
- the large model uses the budget only at train-time

Test Time Scaling Law
and
Comparison of using Compute Budget at Train-time (big model) vs Compute-time (small model)
as the difficulty of a problem varies

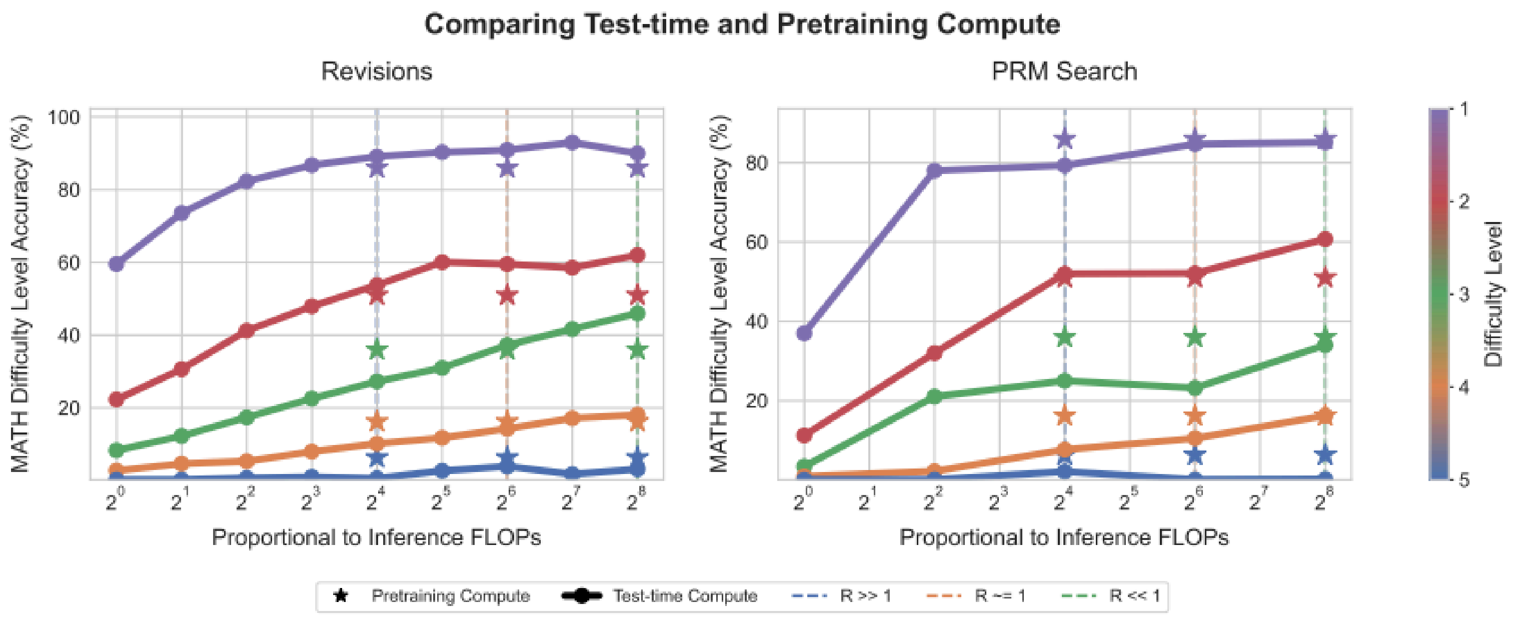


Figure 9 | ***Tradeoff between pretraining and test-time compute in a FLOPs-matched evaluation.*** Each line represents the performance of scaling test-time compute with our compute-optimal policy in each oracle difficulty bin. We plot the results for revisions on the left and search on the right. The stars represent the greedy pass@1 performance of a base model pretrained with ~ 14 times more parameters. We plot test-time compute budget on the x-axis, and place the stars at three different locations along the x-axis, each corresponding to the FLOPs equivalent point of comparison between scaling parameters and scaling test-time compute for three different inference compute loads (e.g. $R = \frac{D_{\text{inference}}}{D_{\text{pretrain}}}$). If the star is below the line, this implies that it is more effective to use test-time compute than to scale model parameters, and if the star is above the line this implies that scaling parameters is more effective. We see that on the easy questions or in settings with a lower inference load (e.g. $R << 1$), test-time compute can generally outperform scaling model parameters. However, on the harder questions or in settings with a higher inference load (e.g. $R >> 1$), pretraining is a more effective way to improve performance.

Interpeting the chart

Problem instances are assigned "difficulty bins"

- each trace represents one level of difficulty
- difficulty represented by colors
    - lower traces are the *harder* problems
    - purple: easy
    - blue: hard

For each difficulty level (color)

- the trace with the solid line represents the small model (test-time compute)
- the stars represent the big model (train-time compute)

So the two traces of the same color

- show the trade-off between Performance and Compute of the test-time compute model

When the horizontal stars of one color

- are below the trace of the same color
- spending the compute budget on test-time yields better Performance than spending it at train-time

For the Revisions method (left chart)

- test-time compute yields better Performance
- for the two *least difficult* classes of examples (purple and red)
- but as difficult level increases (lower traces)
    - big models with more train-time compute
    - perform better

To summarize

- there is a log-linear relationship between Performance and Compute
- which is worthwhile for *less difficult* problems
    - but for *more difficult* problems:
        - larger models (with more train-time compute) are still needed

Strategies for using test-time compute

The above shows the potential for improving Performance by using test-time compute

We examine several common strategies that can be used at test-time to improve Performance.

They fall into two broad classes

- Search
- Reasoning

Search

We use the term "search" to describe

- strategies that explore the space of possible answers

These approaches use an external control process

- to invoke the LLM multiple times
- using logic to control the input to each invocation of the LLM

The model is

- used as a "subroutine" by the external "control" process
- rather than being trained to perform the search directly

Given that a correct answer to our prompt exists: how can we attempt to find it ?

The following chart illustrates some strategies

- to be discussed in subsequent sub-sections

Search Strategies

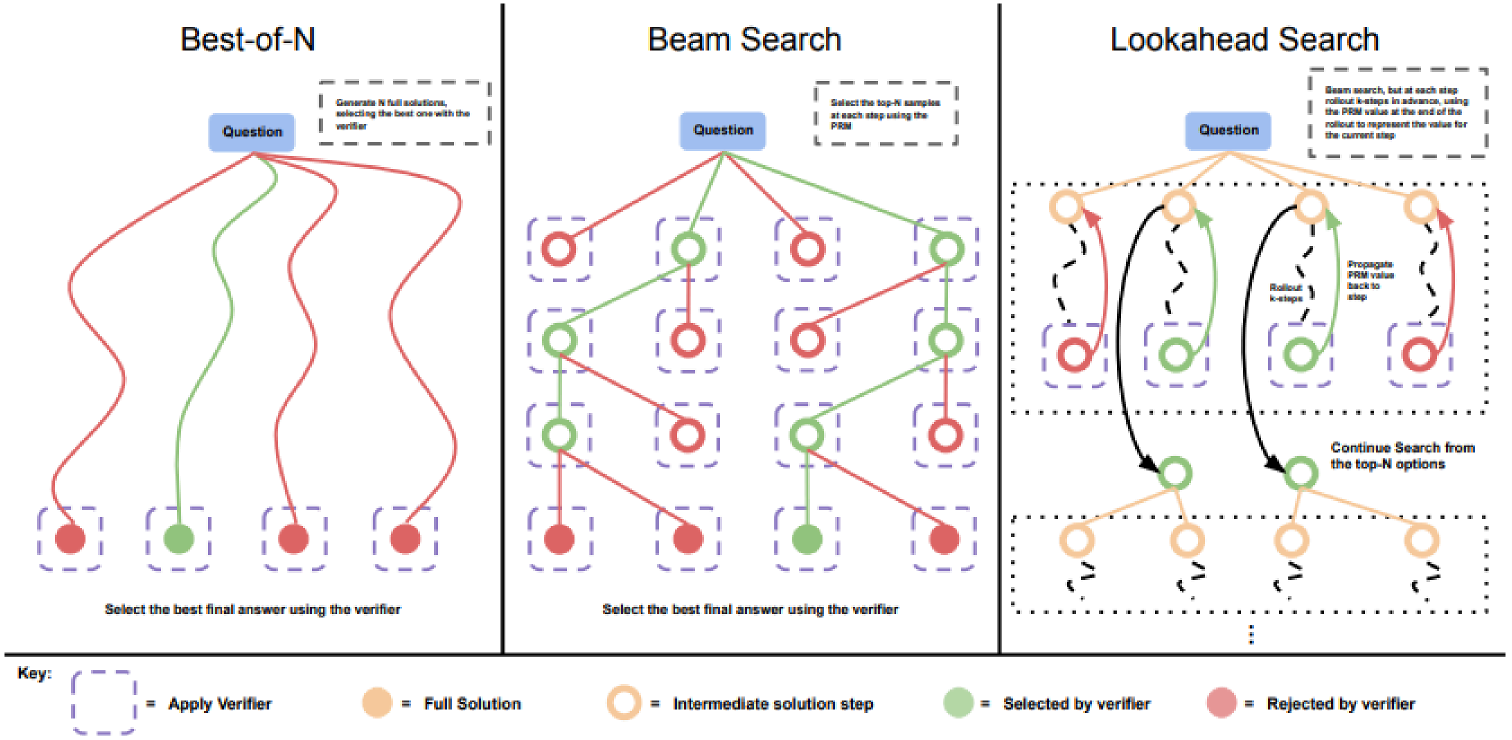

Best-of-N
Beam Search
Lookahead Search
Question
Generate N full solutions, selecting the best one with the verifier
Question
Select the top-N samples at each step using the PRM
Question
Beam search, but at each step rollout k-steps in advance, using the PRM value at the end of the rollout to represent the value for the current step
Rollout k-steps
Propagate PRM value back to step
Continue Search from the top-N options
Select the best final answer using the verifier
Select the best final answer using the verifier
Key:
= Apply Verifier
= Full Solution
= Intermediate solution step
= Selected by verifier
= Rejected by verifier

Figure 2 | ***Comparing different PRM search methods.*** **Left:** Best-of-N samples N full answers and then selects the best answer according to the PRM final score. **Center:** Beam search samples N candidates at each step, and selects the top M according to the PRM to continue the search from. **Right:** lookahead-search extends each step in beam-search to utilize a k-step lookahead while assessing which steps to retain and continue the search from. Thus lookahead-search needs more compute.

Attribution: https://arxiv.org/pdf/2408.03314v1#page=8

Some strategies result in multiple candidate solutions.

How do we decide the single final answer ?

That too will be discussed after presenting the strategies for generating the candidates.

Parallel search

Parallel search

- independently samples multiple answers
    - run the non-deterministic LLM multiple times
    - starting from the same initial state (e.g., random seed)

The left-most example in the chart illustrates this approach.

Systematic search

Rather than sampling complete answers independently

- one can imagine a systematic search of the answer space
- generating intermediate (non-final) answers
- deciding which intermediate answers to improve/which to kill

This is a type of parallel approach, where the answers are not independently generated.

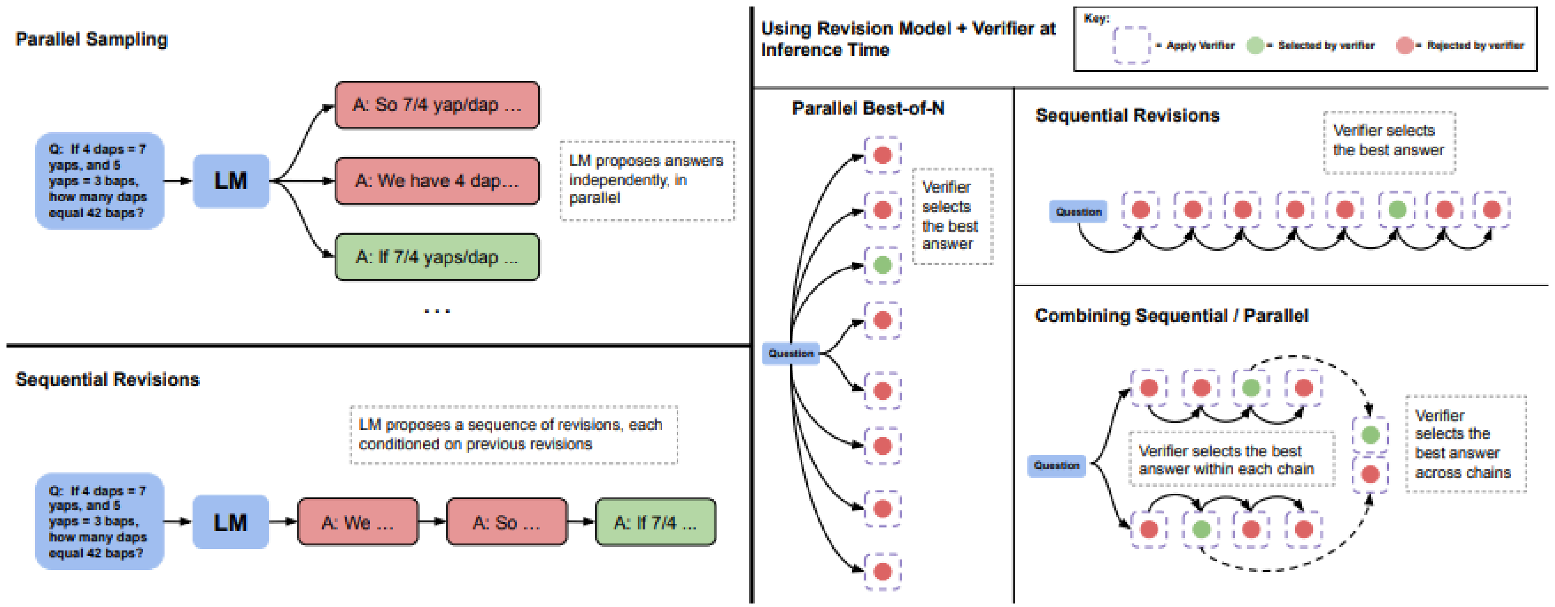
This results in a tree-like structure

- where the leaves are the candidate final answers

The center and right-most example in the chart illustrates this approach.

Sequential search via Revision

Revision strategies



Parallel Sampling
Q: If 4 daps = 7 yaps, and 5 yaps = 3 baps, how many daps equal 42 baps?
LM
A: So 7/4 yap/dap ...
A: We have 4 dap...
A: If 7/4 yaps/dap ...
...
LM proposes answers independently, in parallel
Sequential Revisions
LM proposes a sequence of revisions, each conditioned on previous revisions
Q: If 4 daps = 7 yaps, and 5 yaps = 3 baps, how many daps equal 42 baps?
LM
A: We ...
A: So ...
A: If 7/4 ...
Using Revision Model + Verifier at Inference Time
Key:
= Apply Verifier
= Selected by verifier
= Rejected by verifier
Parallel Best-of-N
Question
Verifier selects the best answer
Sequential Revisions
Verifier selects the best answer
Question
Combining Sequential / Parallel
Question
Verifier selects the best answer within each chain
Verifier selects the best answer across chains

Figure 5 | ***Parallel sampling (e.g., Best-of-N) verses sequential revisions.*** **Left:** Parallel sampling generates N answers independently in parallel, whereas sequential revisions generates each one in sequence conditioned on previous attempts. **Right:** In both the sequential and parallel cases, we can use the verifier to determine the best-of-N answers (e.g. by applying best-of-N weighted). We can also allocate some of our budget to parallel and some to sequential, effectively enabling a combination of the two sampling strategies. In this case, we use the verifier to first select the best answer within each sequential chain and then select the best answer accross chains.

Attribution: https://arxiv.org/pdf/2408.03314v1#page=11

Sequential revisions (above chart lower left and upper right)

Here we use the LLM multiple times *sequentially* to generate a final answer

- the i^{th} answer is critiqued by the LLM
- and used to create a *revised* successor answer $i+1$
    - the successor is not necessarily "better", just a result of responding to the critique

Note

- "parallel" revision is just "Best-of-N"

Reducing multiple candidate answers to a single final answer

Some search strategies result in multiple candidate answers.

There are several strategies for reducing to a single, final answer.

Majority vote

Generate N candidates

- select the candidate that occurs most frequently among the N candidates

Using ranking

The following strategies require the ability to rank the candidates

- may be a partial order, rather than a total order

The section on Verifiers expands on this idea.

For now: we show how ranking may be used to reduce the set of candidates.

Best-of-N

Generate N candidates

- select the single "best" candidate as the final answer

Beam Search

This adapts the search to incorporate the ability to rank *partial* answers.

Generate N candidates at each step

- Select the M (the *beam width*) best answers to continue to explore
- Repeat with the M surviving candidates

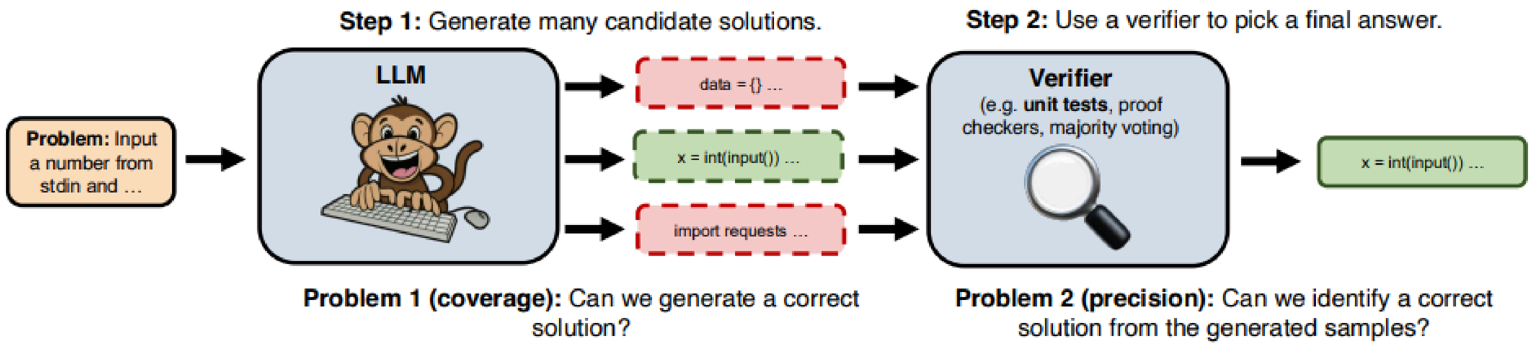
Verifiers: Identifying the ”best” candidate

Given a collection of candidates, how do we identify the best ?

For certain tasks:

- We create a mechanism called a *Verifier*.

Sampling answers



Step 1: Generate many candidate solutions.
Step 2: Use a verifier to pick a final answer.
Problem: Input a number from stdin and ...
LLM
data = {} ...
x = int(input()) ...
import requests ...
Verifier
(e.g. **unit tests**, proof checkers, majority voting)
x = int(input()) ...
Problem 1 (coverage): Can we generate a correct solution?
Problem 2 (precision): Can we identify a correct solution from the generated samples?

Figure 1: The repeated sampling procedure that we follow in this paper. 1) We generate many independent candidate solutions for a given problem by sampling from an LLM with a positive temperature. 2) We use a domain-specific verifier (ex. unit tests for code) to select a final answer from the generated samples.

Attribution: https://arxiv.org/pdf/2407.21787#page=3

For certain tasks

- it is difficult to *efficiently* discover an answer (i.e., not brute-force search)
- but easy to verify whether a candidate is correct

For example

- it is expensive to find Prime numbers
- but easy to verify whether a candidate is Prime

When Verification is possible

- rank candidates with correct answers higher than candidates with incorrect answers

But among multiple correct candidates, a *subjective* mechanism may be needed to choose

- e.g., for a task requiring the construction of a proof of a Theorem
- a shorter correct proof may be more desirable than a longer (but still correct) proof

Limits on Verification

The use of Verifiers is limited by the task

- Math problems are often verifiable
    - test the correctness of the answer
- Text problems are often *not verifiable*
    - Performance is subjective
    - human preference as the which is the "best" answers
        - e.g., which summary of a long academic paper is best ?

Reward models

For tasks where Verification is not possible (or results in ties)

- we train a Neural Network to rank candidates
- generating a "reward" for each candidate.

These models are trained using examples created by Human Feedback

- Human prompts an LLM to generate multiple answers to a problem
- Human ranks the answers (really: creates pairs of candidates and identifies the preferred one)
- Neural Network Classifier trained to identify the preferred candidate in the pair

A verifier which is used to verify/rank only *complete* candidate answers

- is called an *Output Reward Model (ORM)*

These can be used to rank the multiple candidates as required by some of the above strategies for producing candidate answers.

More sophisticated reward models

- can be used to *guide* the search
- by evaluating each step in the generation of a single candidate answer.

Such models are called *Process Reward Models (PRM)*

These are commonly used when the answers contain *chains of thought/reasoning*

- a PRM can assign a score to *each reasoning step*
    - can terminate a branch of the search upon discovering an incorrect reasoning step
    - can choose which branches to extend by ranking the quality of the reasoning
        - can be combined with back-tracking

A candidate with a fatally incorrect reasoning step has a low reward.

A candidate with a correct answer has a higher reward

- can combine the per-step rewards into a final reward/score for the candidate
- choose as final answer the candidate with highest reward

Reasoning

The second approach

- trains a model
- to use a "Chain of Thought" solution strategy
    - producing a final answer
    - as the culmination of a process of "thinking step by step"
    - that culminates in a final answer

The model produces the answer on its own

- no external control process
- using solution strategies that it has been trained (via examples) to use

Reasoning can also involve *strategies* that lead to better solutions

Some useful strategies

- *revision*
    - abandon an intermediate solution
    - change the intermediate solution and extend
- *reflection*
    - "reflect on" (i.e., evaluate)
    - intermediate solutions (reasoning traces that have not yet created an answer)
    - to determine whether the reasoning trace is on "the right track" to a good solution

Combining training-time and test-time methods for increasing Performance

In theory

- we could train the model to use long Chain of Thought reasoning traces
- use the reasoning model as a "subroutine" in a test-time process as in Test-time scaling

Strategies such as Reflection and Revision

- could be implemented as part of the test-time control program
- or *instilled into the model* as part of training

```
In [2]: print("Done")
```

```
Done
```