# To Backtrack or Not to Backtrack: When Sequential Search Limits Model Reasoning

**Tian Qin***
Harvard University

**David Alvarez-Melis‡**
Harvard University, Kempner Institute, MSR

**Samy Jelassi‡**
Harvard University, Kempner Institute

**Eran Malach‡†**
Harvard University, Kempner Institute

## Abstract

Recent advancements in large language models (LLMs) have significantly improved their reasoning abilities, particularly through techniques involving search and backtracking. Backtracking naturally scales test-time compute by enabling sequential, linearized exploration via long chain-of-thought (CoT) generation. However, this is not the only strategy for scaling test time-compute: parallel sampling with best-of-$n$ selection provides an alternative that generates diverse solutions simultaneously. Despite the growing adoption of sequential search, its advantages over parallel sampling—especially under a fixed compute budget—remain poorly understood. In this paper, we systematically compare these two approaches on two challenging reasoning tasks: CountDown and Sudoku. Surprisingly, we find that sequential search underperforms parallel sampling on CountDown but outperforms it on Sudoku, suggesting that backtracking is not *universally* beneficial. We identify two factors that can cause backtracking to degrade performance: (1) training on fixed search traces can lock models intro suboptimal strategies, and (2) explicit CoT supervision can discourage 'implicit' (non verbalized) reasoning. Extending our analysis to reinforcement learning (RL), we show that models with backtracking capabilities benefit significantly from RL fine-tuning, while models without backtracking see limited, mixed gains. Together, these findings challenge the assumption that backtracking universally enhances LLM reasoning, instead revealing a complex interaction between task structure, training data, model scale, and learning paradigm.

## 1 Introduction

Recent studies (Kumar et al., 2024; Havrilla et al., 2024) propose teaching LLMs to correct mistakes through *backtracking*, enabling exploration of alternative solutions. Despite growing popularity (DeepSeek-AI et al., 2025; Muennighoff et al., 2025), it remains unclear whether correcting errors post-hoc via backtracking is ultimately more compute-efficient at test time than directly learning the correct solution. Solving strategic games such as CountDown and Sudoku requires extensive exploration of different solution paths, making them ideal for analyzing the computational trade-offs of sequential versus parallel search. In this work, we use these two games to conduct a controlled investigation to determine whether backtracking is an effective way to scale test-time compute.

There are two primary strategies to scale LLMs' test-time compute: sequential autoregressive search (explicit backtracking within a chain-of-thought) and parallel sampling (generating multiple independent solutions and selecting the best with best-of-$n$). While sequential search allows the model to refine reasoning by learning from past mistakes, it comes at a cost: due to the attention mechanism, the FLOPs required to generate CoT grow quadratically

*Correspondence to tqin@g.harvard.edu ‡ Equal senior contributions.
†Currently at Apple.

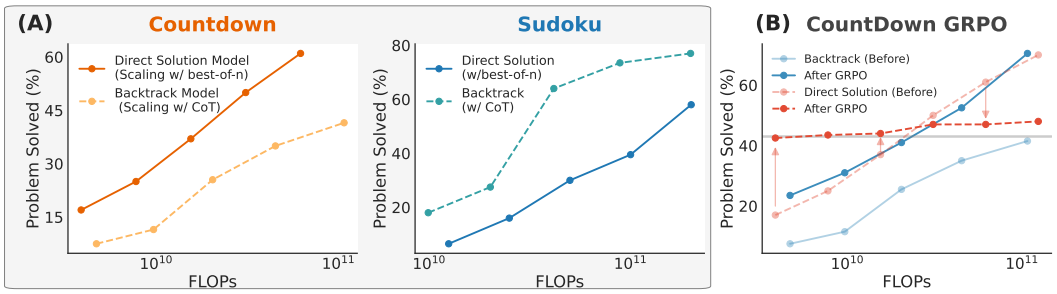


Figure 1: **Backtracking performance varies significantly with task type and the application of post-training reinforcement learning.** *(A)* Training backtracking and direct solution models on CountDown and Sudoku reveals task-dependent performance: under equal test-time compute, backtracking (sequential search) underperforms direct solution with best-of-$n$ generation (parallel search) on CountDown, but outperforms it on Sudoku. *(B)* Fine-tuning with GRPO consistently improves backtracking model performance across compute budgets, but has mixed effects on the direct solution model.

with sequence length. Even when generating the same number of tokens, sequential search incurs more FLOPs than parallel sampling. To compare these two strategies, we train (i) **backtracking models** that learn from explicit search traces and use sequential search to solve hard problems, and (ii) **direct solution (i.e., no backtracking) models** that learn solely from correct solutions, using parallel search at test time. Equating test-time compute, we observe contrasting results (Fig. 1 *A*): in CountDown, the backtracking model consistently **underperforms**, whereas in Sudoku, it consistently **outperforms** the direct solution model.

Through controlled experiments, we identify two reasons teaching backtracking can inadvertently degrade performance. First, explicit backtracking reasoning traces bias models toward prescribed search strategies, **limiting exploration** of potentially superior alternatives. In CountDown, the backtracking model closely mimics training search paths, while the direct solution model independently discovers more efficient strategies (Section 4.2). Second, detailed backtracking traces **encourage verbosity** (producing lengthy yet ineffective reasoning chains), while discouraging internal "thinking" (implicit reasoning without outputting CoT, Section 4.3). Beyond these factors, we demonstrate that model size and task-specific characteristics also impact the effectiveness of backtracking (Section 5.1). Crucially, we show that our contrastive observation between Sudoku and Countdown **generalizes** to real-world tasks: such as math and science problem solving. We show that backtracking is *not* always the most effective way to scale test-time compute (Appendix A) for general reasoning models.

Extending beyond supervised learning, we evaluate reinforcement learning (RL) with Group Relative Policy Optimization (GRPO) (Shao et al., 2024), uncovering novel interactions between backtracking capabilities and RL. We show that the backtracking model discovers new, effective search strategies through RL, achieving substantial performance improvements. Conversely, the direct solution model improves one-shot accuracy but loses effectiveness in parallel search, revealing a clear trade-off (Fig. 1 *B*). This finding shifts our understanding of how backtracking influences a model's potential to improve under RL, highlighting the unique advantage of teaching backtracking for long-term reasoning capabilities.

Our controlled study on two strategic games provides a nuanced understanding of when backtracking effectively scales test-time compute. Our main contributions are:

- We use CountDown and Sudoku as controlled testbeds to examine whether backtracking enables efficient test-time scaling. Under a fixed compute budget, backtracking outperforms parallel search in Sudoku but underperforms in CountDown (Fig. 1 *A*).
- We identify two key factors affecting backtracking efficacy: (1) **Prescribed search bias**: Training on detailed backtracking traces can unintentionally constrain models to subopti-

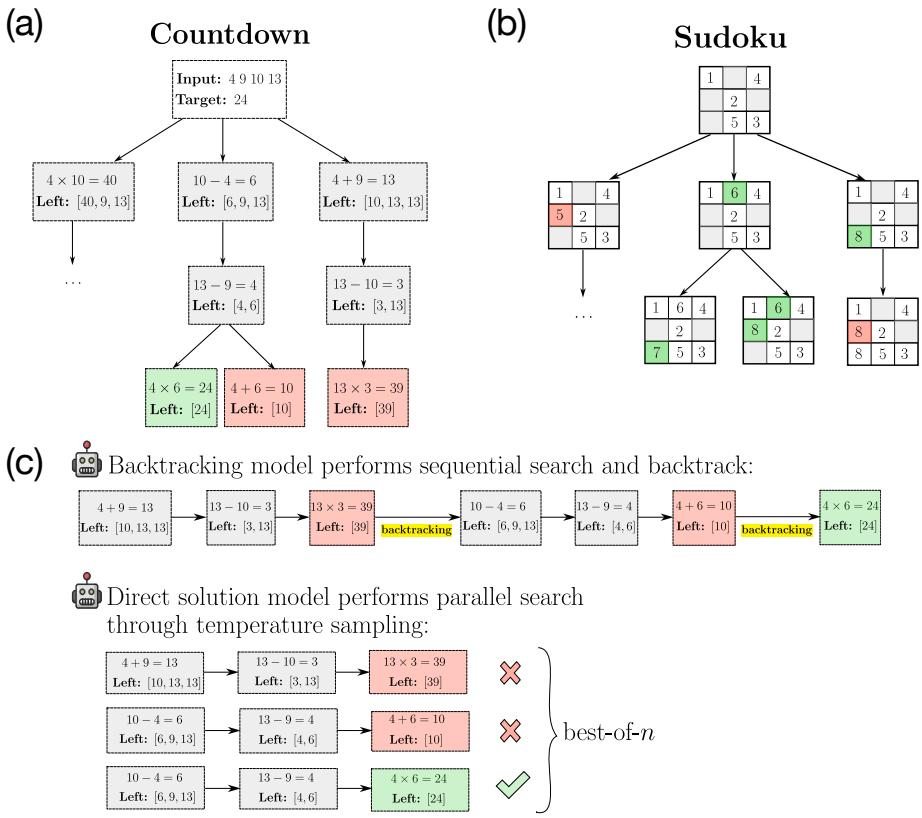


Figure 2: **Backtracking and direct solution for two different strategic games.** *Panel (a, b):* Example the search tree for CountDown and Sudoku. Solving both games require extensive search in the solution space. *Panel (c):* The backtracking model is trained on the search traces generated by a Depth-First-Search (DFS) algorithm. At test time, the model performs sequential search. The direct solution model is trained on the correct solution only. At test time, the model performs parallel search through temperature sampling and takes best-of-$n$.

mal search strategies. (2) **Excessive verbosity:** Explicit backtracking traces encourage models to produce lengthy reasoning chains without improving reasoning ability.

- We demonstrate that reinforcement learning (GRPO) consistently enhances backtracking models by enabling discovery of novel solutions, whereas direct solution models experience mixed outcomes (Fig. 1 *B*).

## 2 Related Work

See Appendix B for an extensive review on related work.

**Scaling test-time compute .** Prior work has explored scaling language model performance at test time through parallel or sequential search strategies. Parallel methods rely on independent sampling and selection via heuristics or reward models (Brown et al., 2024; Irvine et al., 2023; Levi, 2024; Xin et al., 2024), while sequential methods refine reasoning step by step using earlier outputs (Hou et al., 2025; Lee et al., 2025). Tree-based methods such as MCTS bridge the two and often incorporate process-level reward models to guide reasoning (Wu et al., 2024; Lightman et al., 2023). Our work contributes to this area by comparing sequential (backtracking) and parallel search under fixed compute budgets.

**Self-correction and backtracking.** Language models can be trained to self-correct through fine-tuning on revision data, synthetic augmentations, or reward-based learning (Saunders et al., 2022; Qu et al., 2024; Welleck et al., 2022). Some approaches also introduce explicit search or separate correction modules to guide revision (Yao et al., 2023b; Havrilla et al.,

2024). We build on this line of work by studying backtracking as an implicit form of self-correction, analyzing when learning to backtrack helps or hinders reasoning.

**Reinforcement learning for LLM reasoning.** Reinforcement learning has shown promise in enabling language models to autonomously discover reasoning strategies, including through simplified algorithms like GRPO (Shao et al., 2024; DeepSeek-AI et al., 2025). While prior work has demonstrated strong results, it remains unclear which model properties enable successful RL-based reasoning (Zelikman et al., 2022; Kazemnejad et al., 2024). Our study addresses this gap by comparing how backtracking and no backtracking models respond to RL fine-tuning, revealing asymmetric benefits.

## 3 Two strategic games: CountDown and Sudoku

### 3.1 CountDown

#### *3.1.1 Game setup*

The Game of CountDown has been frequently used as a testbed to study and evaluate LLM reasoning (Gandhi et al., 2024; 2025; Yao et al., 2023a). In a CountDown game, the player is given a set of candidate numbers and a target number (restricted to integers). The goal is to reach the target by applying a sequence of arithmetic operations—addition, subtraction, multiplication, or division—using the candidate numbers. Each number must be used exactly once, and intermediate results can be reused in subsequent operations.

To algorithmically solve CountDown, we can represent the problem as a search tree (Fig. 2*a*). Each node in the search tree corresponds to a state defined by the current set of available numbers. At each step, the algorithm selects a pair of numbers from the set and applies one of the four operations, replacing the pair with the resulting value to create a new state. This process continues recursively until the target number is reached (correct leaf node) or all combinations are exhausted (wrong leaf node). In this work, we play the CountDown with four candidate numbers, and for each game, there are 1,152 possible search paths.

#### *3.1.2 Data generation*

We generate **backtracking traces** with Depth First Search (DFS) with a sum-heuristic (Gandhi et al. (2024), further details in Appendix C.1). We generate a dataset of 500,000 CountDown questions, and the DFS search correctly solves 57% of the questions. The backtracking trace is a serialized version of DFS, listing all the tree nodes visited in the order of DFS traversal. To construct the **direct solution** training data, we prune the backtracking traces to keep only the correct solution path. With the pruning approach, we remove the exploratory parts of the trace while preserving the answer format and scaffolding used in the backtracking model, to ensure a fair comparison. We also ensure that the direct solution model does not see more solved CountDown games, we include only the 285,000 questions (i.e., 500,000 $\times$ 0.57) that DFS successfully solves. We provide examples of both training data in Appendix H.

### 3.2 Sudoku

#### *3.2.1 Game setup*

Sudoku is another prototypical strategic game used to study reasoning and search in LLMs (Yao et al., 2023a; Long, 2023). In this work, we focus on hard $9 \times 9$ Sudoku boards, where only about 20 of the 81 cells are pre-filled, making the search space substantially larger (see Appendix C.1 for a description of Sudoku rules). To algorithmically solve Sudoku, we represent the problem as a search tree (Fig. 2*b*). Each node corresponds to a partial board state, where some cells have been filled. At each step, the algorithm selects an unfilled cell and fills it with a candidate digit that satisfies Sudoku constraints in the current state. Each valid assignment creates a new child node representing the updated board. The process continues recursively until a complete, valid solution is reached (correct leaf node) or no valid moves remain (wrong leaf node). The depth of the tree corresponds to the number of empty cells, and the branching factor at each node depends on the number of unfilled cells as well as how many digits are valid for each unfilled cell.

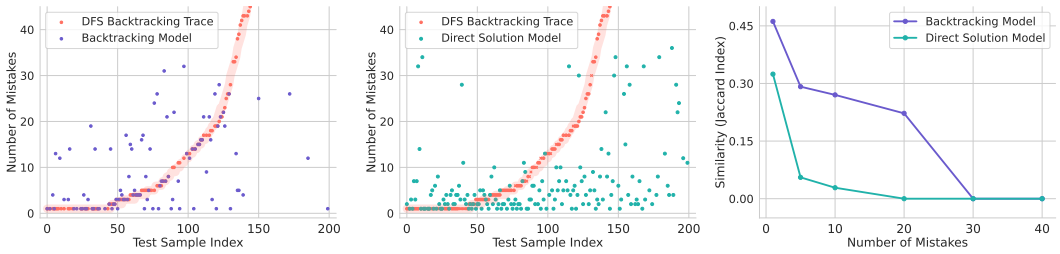


Figure 3: **Backtracking and direct solution models implement different search strategies for CountDown.** For test questions that model solves correctly, we measure the number of mistakes made (i.e., wrong terminal nodes visited) before finding the correct solution. We sort the test questions by number of mistakes made by DFS. *Left:* Trained on DFS traces, the number of mistakes made by the backtracking model correlates with the DFS. *Middle:* In contrast, the direct solution model solves a lot more problems with significantly fewer mistakes compared to DFS. *Right:* For a given number of mistakes made, we examine whether two models solve the same set of question as DFS. Direct solution model implements a search strategy significantly different from DFS.

#### 3.2.2 *Data generation*

We follow the same procedure as CountDown to generate training data for both the backtracking and direct solution models. We use a DFS-based search algorithm, in combination with a Sudoku solver that applies seven common human strategies (e.g., naked singles, hidden pairs and etc, Papadimas. (2023) ) to eliminate candidates for unfilled cells. At each node, we use the 7 strategies to eliminate candidates for unfilled cells, and then DFS chooses an unfilled cell, makes a guess and continues solving recursively. This process continues until the board is either solved or reaches a dead-end (i.e., an invalid state with no legal moves). We use a dataset of 3M Sudoku puzzles from (Radcliffe, 2020), and the combined DFS-solver approach successfully solves 98% of them. Since DFS successfully solves nearly all puzzles, we train both models on 2.8M examples and reserve the last 200K for validation and testing. We provide further details on Sudoku training data generation in Appendix C.1 and data examples in Appendix H.

### 3.3 Model and training

We use Qwen2.5-style model architectures (Yang et al., 2024) with RoPE positional encoding (Su et al., 2021) and Group Query Attention (GQA) (Ainslie et al., 2023). To maximize parameter efficiency, we design custom tokenizers for both games, significantly reducing the size of the language modeling head. This allows us to train smaller models than prior work (Gandhi et al., 2024; Shah et al., 2024) while maintaining comparable performance on both tasks. For CountDown, we use a 17M parameter model with a context length of 4096 tokens; for Sudoku, we use a 38M model with the same context length. See Appendix C.3 for model architecture and an exhaustive list of training hyperparameters. We train all models until validation loss converges (see Appendix G.3).

## 4 Empirical trade-offs of backtracking

We first demonstrate that backtracking models do not universally outperform the direct solution models (Section 4.1) because backtracking models are restricted to learn a prescribed way of search (Section 4.2). We then identify two factors (Sections 4.3) showing how we might improve test-time scaling for backtracking models.

### 4.1 Backtracking is not always beneficial

**Evaluation metrics.** We evaluate model performances using solving accuracy on 200 unseen problems with binary scores (either correct or incorrect, no partial credits, see appendix C.1). We use FLOPs to compare inference costs (see Appendix D for FLOPs computation). For the backtracking model, we allow models to autoregressively generate and measure how many problems the model finds the correct solution at various CoT lengths (ranging from 1024 to 4096 tokens). For the direct solution model, we generate

$n$ solutions in parallel through temperature sampling at $T = 0.7$, and examine whether the model has found the correct solution within $n$ attempts (i.e., best-of-$n$). Best-of-$n$ is a suitable choice in those two games, a case where solving the task is hard but verification is trivial. In general, our analysis applies to tasks where verification can be easily done with an external verifier at test-time. This is definitely not always the case, and we leave the study of problems where test-time verification is not as easy to future work. In those tasks, one might need to consider majority voting or other strategies. See Appendix E for further discussions.

**Results.** In Fig. 1 *A*, we observe distinct scaling behaviors for the two models. For both games, the direct solution model's test accuracy scales *linearly* with increased test-time compute (measured on a logarithmic scale). This scaling behavior indicates that through parallel sampling, the backtracking model generates diverse solutions that search through different solution paths. Conversely, the backtracking model exhibits *sub-linear* scaling: Longer solution traces disproportionately yield smaller accuracy improvements. We attribute the sub-linear scaling to two causes. First, as reasoning chains become longer, the backtracking model might struggle to effectively track visited states and efficiently search through the solution space. Second, when models perform sequential search, the computation cost grows quadratically with CoT length (due to the attention mechanism, see Appendix D), and this further makes backtracking model less effective for scaling up test time compute. Overall, for CountDown, the direct solution model consistently outperforms its backtracking counterpart. However, this trend is reversed in Sudoku, where the backtracking model consistently achieves higher accuracy.

### 4.2 Backtracking model learns both the good and the bad

When teaching a child to correct math mistakes, the child understands that the goal is the correct answer—not making and then fixing errors. Humans have meta-cognitive awareness that models lack. Models trained via next-token prediction simply imitate the traces they see, including making the mistake before fixing it. In CountDown, this poses a key limitation: the backtracking model learns to follow the specific search paths seen in training. While some tasks—like shortest path finding—have optimal strategies we can supervise directly (e.g., Dijkstra's algorithm), most reasoning tasks, including CountDown, lack such guarantees. As a result, the model may be constrained by the inefficiencies in the backtracking data. In contrast, the direct solution model, trained only on correct answers, is free to discover more efficient strategies. In our subsequent analysis, we concretely show how the direct solution model successfully bypasses many inefficient search and backtracking steps learned by the backtracking model.

#### *4.2.1 Backtracking model finds the solution with fewer mistakes*

**Measuring number of mistakes.** We compare the number of mistakes made by: (1) DFS (used to generate backtracking data), (2) the backtracking model, and (3) the direct solution model. For DFS and the backtracking model, mistakes are counted as the number of incorrect terminal nodes explored before finding the correct solution. For the direct solution model, mistakes correspond to how many parallel samples ($n$ in best-of-$n$) are needed. [1]

**Comparing search strategies.** We sort the 200 test problems based on mistakes made by DFS and plot mistakes for both models. Fig. 3 *left* compares DFS search and backtracking model. The number mistakes made by the backtracking model is correlated with the DFS backtracking trace. This observation is not surprising given that the backtracking model is trained on these traces. However, this result is interesting when we compare it against the direct solution model (Fig. 3 *middle*). The direct solution model solves most problems within fewer than 10 attempts—far fewer compared to DFS or the backtracking model. Fig. 3 *right* quantifies these observations. Specifically, for a fixed mistake budget, we use Jaccard Index to measure whether the model solves a similar set of problems as DFS solves. The backtracking model closely mirrors DFS search (high set similarity), whereas the direct solution model diverges significantly (low set similarity). Together with superior performance of the direct solution model, we conclude that the direct solution model learns more efficient search strategies, avoiding unnecessary explorations of wrong paths.

[1] Mistakes are counted only for problems solved correctly by the model.

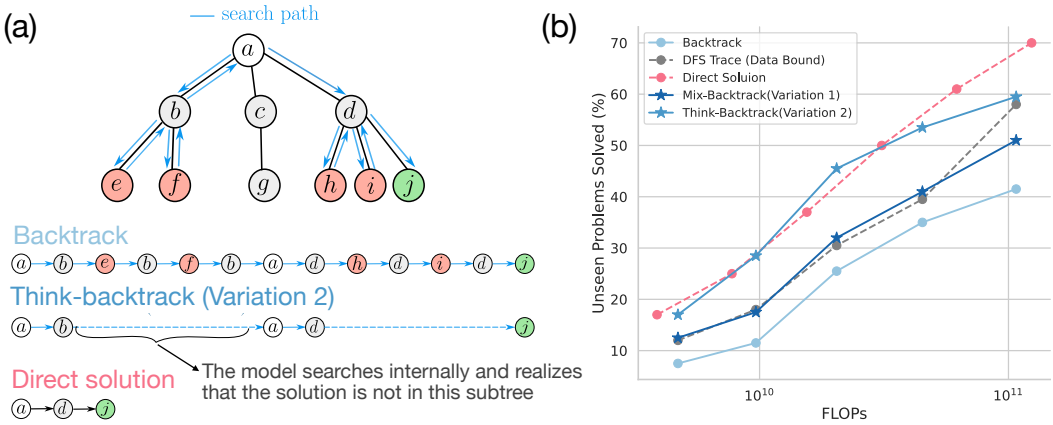


Figure 4: **Two different variations to improve backtracking model.** *(a).* We hypothesize that the backtracking model can think one step ahead without sacrificing its ability to search. Therefore, we shorten the search trace by skipping the last search step. *(b).* Two data variations that improve the backtracking model. Mixed-backtrack model trained on a diverse set of search strategies. Think-backtracking model trained on shortened DFS trace.

### 4.3 Two ways to improve backtracking model

**Training on diverse set of search strategies.** Our analysis suggests a clear direction for improving the backtracking model: using better search strategies to improve backtracking traces. Beyond DFS, we explored alternatives including Breadth-First Search (BFS) and various heuristic methods (see Appendix G.1). Despite these efforts, no single search strategy significantly outperformed DFS. Inspired by Gandhi et al. (2024), we trained a variant of the backtracking model—*mix-backtrack* model—-using a diverse mixture of BFS and DFS strategies (32 in total), aiming to help the model discover more optimal search patterns.

**Backtracking model *thinks less and talks more*.** Apart from learning suboptimal search strategies, another inefficiency in the backtracking model is caused by the model learns to be excessively verbose. Specifically, by requiring the model to explicitly output every step of the DFS, we may prevent it from internalizing part of the reasoning process. Concretely, we hypothesize that for CountDown, the model can internally plan at least one step ahead, allowing it to shorten its explicit reasoning trace without losing its ability to perform DFS. To test hypothesis, we train a variation—the *think-backtrack* model—on shortened DFS traces, skipping one intermediate step (Fig. 4, *A*).

**Mix-strategy results.** Fig. 4 (*B*) compares this mixed-strategy model against the original backtracking and direct solution models. We also include a training data upper bound, representing perfect execution of the mixed search strategies. The mixed-strategy model improves over the original backtracking model and closely approaches its training-data upper bound. However, even with deliberate attempts to optimize search strategies, surpassing the performance of the direct solution model remains challenging. This experiment underscores the inherent difficulty in identifying superior handcrafted search traces.

**Think-backtrack results.** Fig. 4 (*B*) also compares the performance of the think-backtrack model. By encouraging the model to internalize parts of the reasoning process, the think-backtrack model achieves performances comparable to the direct solution model. This result suggests that models with backtracking ability might produce long but ineffective CoT. By training the model to avoid making the mistakes at the first place, we reduce model verbosity without sacrificing its search capability, and in turn improving test-time-compute scaling. As an additional evidence, in Appendix G.2, we show that the think-backtrack model solves a superset of test problems solved by the original backtrack model.

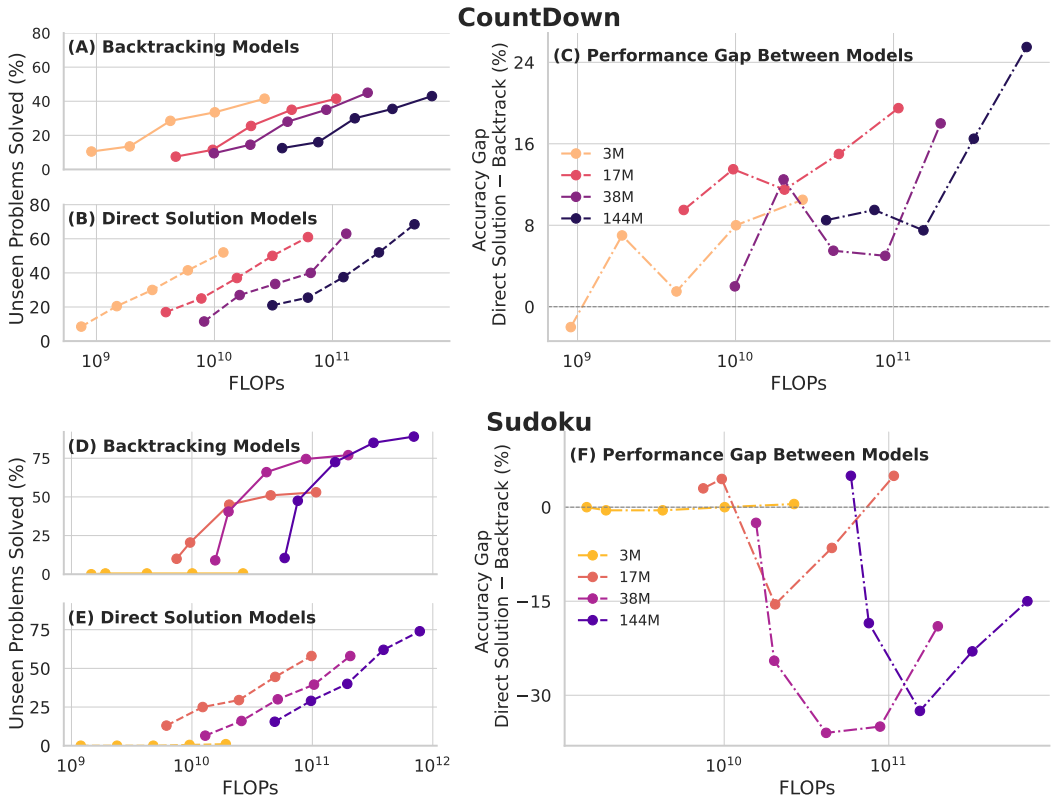


Figure 5: **Different scaling behaviors for backtracking versus direct solution model. CountDown** *(A)*. Backtracking model performance does not improve as we scale up model size. *(B)*. The direct solution model improves *(C)*. Direct solution model consistently *outperforms* backtracking model. **Sudoku** *(D, E)*. Both models' performances improve as we scale up model size. *(F)*. Direct solution model consistently *underperforms* backtracking model.

## 5 Model size and tree depth impact the efficacy of backtracking

While we've shown that backtracking might lead to ineffective test-time scaling, other factors also shape its effectiveness. In Section 5.1, we show that backtracking and direct solution models scale differently with model sizes. To explain the contrasting outcomes (Fig. 1 *A*) between CountDown and Sudoku, in Appendix F, we show that task differences—particularly search tree depth—play a key role: deeper tasks like Sudoku benefit more from backtracking.

### 5.1 Dependence on model size

We now investigate how model size impacts the performance of backtracking and direct solution models. We evaluate four model scales—3M, 17M, 38M, and 144M—by proportionally increasing the number of attention heads, embedding dimensions, and number of attention layers. Detailed model configurations can be found in Appendix C.2.

**CountDown.** Scaling up model size improves the performance of the direct solution model (Fig. 5 *B*) across all test-time-compute budgets. When trained exclusively on correct solutions, larger models can independently discover highly effective search strategies. In contrast, the backtracking model shows *no* improvements with increased model sizes (Fig. 5 *A*). The lack of improvement from model scaling can be explained by training data: The performance of backtracking model is constrained by the quality of the backtracking traces used for training. As previously seen in Fig. 4 (*right*), the 17M backtracking model is already approaching the performance ceiling that is set by the training data. Training larger models on the same backtracking data would not lead to further performance improvements. Due

to different scaling behaviors between backtracking and direct solution models, the gap in performances between two types of models widens with increasing model sizes (Fig. 5 *C*).

**Sudoku.** Similar to CountDown, the performances of direct solution models improve with increased model sizes (Fig. 5 *E*). Unlike CountDown, however, the backtracking model also significantly benefits from scaling (Fig. 5 *D*). This difference can again be explained by examining the backtracking training data. Sudoku is inherently more complex than CountDown. The DFS backtracking traces successfully solve 97% of test boards—far exceeding the current performance of all four tested model sizes. Because the backtracking model for Sudoku has not yet reached training data performance ceiling, increased model capacity leads to improved results. On the other hand, due to the complexity and large search space of the game, the backtracking models' performance gains start to diminish as the search traces become longer. As a result, the backtracking model consistently outperforms the direct solution model across scales, but the advantages diminishes at larger compute budgets (Fig. 5 *E*).

## 6 GRPO: Learning beyond the imitation game

So far, we have shown that under supervised learning, backtracking is not always optimal for scaling test-time compute. We now explore how further training both backtracking and direct solution models with reinforcement learning leads to qualitatively different outcomes.

### 6.1 Continue training models with GRPO

Recently, RL has become a popular approach to further enhance LLMs performance on challenging benchmarks such as MATH (Hendrycks et al., 2021) and AIME (AIME, 2024). Here, we study the effects of RL in a controlled setting, focusing on how it impacts a model's backtracking behaviors (sequential search) and as well as a model's parallel search capability (sampling with best-of-$n$). We take the CountDown backtracking and direct solution models, which have been trained to convergence under the supervised learning objective (see Appendix G.3 for training curves). We then continue training each model using GRPO (Shao et al., 2024), following verl's (Sheng et al., 2024) implementation. We perform GRPO on the same training data used for the supervised learning. As before, we evaluate performance across different test-time compute budgets.

### 6.2 Backtracking model discovers new search strategies

Figure 1 *C* shows that the backtracking model post GRPO sees an performance boost across all test-compute budgets. The post-GRPO model (*dark red*) reaches an accuracy comparable to the pre-GRPO direct solution model (*light blue*). This improvement is surprising for two reasons: (1) at maximum compute (4096 tokens), the model solves nearly 70% of the test set—exceeding the performance of the DFS strategy used to generate training data (57%); and (2) the model was trained on questions it has already seen during supervised learning, with no new problems introduced during GRPO.

These gains suggest that the backtracking model, once freed from the constraints of predicting next token on DFS traces, can now discover better search strategies. To concretely show that the backtracking model post-GRPO learns search strategies different from DFS training traces, we revisit the mistake-counting analysis from Section 4.2.1 (Figure 3). For each test problem, we compute the number of mistakes as before (i.e., counting how many incorrect terminal nodes are explored before reaching a correct solution). Using the same set similarity measure as before, we quantify the strategy deviation in Figure 6 (*left*). The smaller Jaccard index values confirm that through GRPO, the backtracking model has learned new and more effective search behaviors. In Appendix G.4, we also show the per-problem scatter plot as done in Figure 3.

### 6.3 Direct solution model specializes at pass@1

We now show that compared to backtracking models, GRPO has remarkably different effects on direct solution models. As shown in Figure 1 *C*, the direct solution model post-GRPO achieves strong performance at the smallest compute budget (pass@1), solving 42.5% of unseen CountDown puzzles (82 out of 200 test problems). None of the handcrafted search strategies (Appendix G.1) can reach such high accuracy. To understand the impressive gain

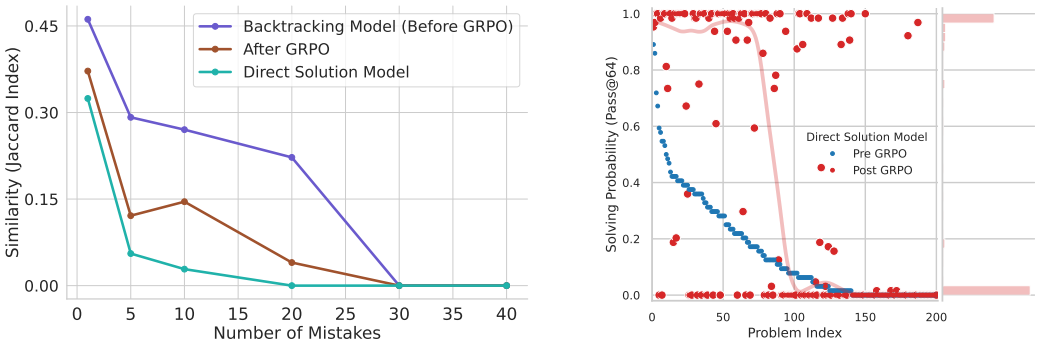


Figure 6: **GRPO has different effect on backtracking versus direct solution model** *Left:* After GRPO, the backtracking model's search strategy starts to deviate away from the DFS search. *Right:* For problems the pre-GRPO direct solution model (*blue*) have a non-zero pass@$k$ solving probabilities, the post-GRPO direct solution model (*red*) solves with pass@1.

on 1-shot performance, we examine those 82 problems, and discover that the pre-GRPO direct solution model was able to find correct solution by sampling best-of-$n$ (with $n \leq 64$). We now examine a model's solving probabilities (i.e., measuring pass@$k$ rate out of the 64 generations). We compare the pass@$k$ rate for the diret solution model pre and post GRPO, shown in Figure 6, *right*. We rank the 200 test problems by the pre-GRPO model's solving probabilities. For problems that the pre-GRPO model has a non-zero pass@k rate, the post-GRPO model can solve most of them with pass@1.

However, this improvement in 1-shot performance comes with a substantial trade-off: the model loses its ability to generate diverse solutions. As a result, when we perform parallel search using best-of-$n$, the direct solution model post-GRPO fail to explore different solution paths, hurting its test-time-scaling effectiveness. Therefore, test-time compute scaling becomes ineffective as we increase compute bugdets, forming a sharp contrast to the backtracking model's consistent improvements across the full compute budget.

## 7 Conclusion and discussions

In this work, we conducted a controlled empirical investigation into the efficacy of teaching backtracking to large language models (LLMs) as a method for scaling test-time computation. Using two strategic games, CountDown and Sudoku, we demonstrated that backtracking does not universally outperform parallel solution strategies; rather, its effectiveness depends significantly on task characteristics, model scale, and training approach. Appendix A, we show that our resuls in synthetic setting generalize: even in real-world reasoning tasks, backtracking is *not* always beneficial. Additionally, our reinforcement learning experiments uncovered a unique synergy between backtracking capabilities and RL-based training, enabling models to discover novel strategies.

**Limitations and future work.** While our experiments relied on two strategic games (CountDown and Sudoku) and models trained from scratch—common practices for controlled studies—an important avenue for future research is extending our findings to complex, real-world reasoning tasks such as coding and mathematical problem-solving. For future work, developing precise ways to characterize tasks that benefit from backtracking will be valuable for guiding model training. Finally, while we intentionally created a dichotomy between pure backtracking and direct-solution models, real-world applications may require hybrid strategies that dynamically choose between direct generation and explicit search based on problem complexity. Investigating whether LLMs can autonomously optimize their reasoning modes, particularly through reinforcement learning paradigms, is a promising future direction for improving the flexibility and efficiency of model reasoning.

## Acknowledgments

We thank Core Francisco Park and Bingbin Liu for helpful discussions and feedback throughout the development of this work. TQ and DAM acknowledge support from the Kempner Institute, the Aramont Fellowship Fund, and the FAS Dean's Competitive Fund for Promising Scholarship.

## References

AIME. American invitational mathematics examination, 2024. URL https://maa.org/math-competitions/american-invitational-mathematics-examination-aime.

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv [cs.CL]*, 2023. URL http://arxiv.org/abs/2305.13245.

Afra Feyza Akyürek, Ekin Akyürek, Aman Madaan, Ashwin Kalyan, Peter Clark, Derry Wijaya, and Niket Tandon. RL4F: Generating natural language feedback with reinforcement learning for repairing model outputs. *arXiv [cs.CL]*, 2023. URL http://arxiv.org/abs/2305.08844.

Zachary Ankner, Mansheej Paul, Brandon Cui, Jonathan D Chang, and Prithviraj Ammanabrolu. Critique-out-loud reward models. *arXiv [cs.LG]*, 2024. URL http://arxiv.org/abs/2408.11791.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv [cs.LG]*, 2024. URL http://arxiv.org/abs/2407.21787.

Sehyun Choi, Tianqing Fang, Zhaowei Wang, and Yangqiu Song. KCTS: Knowledge-constrained tree search decoding with token-level hallucination detection. *arXiv [cs.CL]*, 2023. URL http://arxiv.org/abs/2310.09044.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z F Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J L Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R J Chen, R L Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S S Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W L Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X Q Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y K Li, Y Q Wang, Y X Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y X Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z Z Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie,

Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv [cs.CL]*, 2025. URL http://arxiv.org/abs/2501.12948.

Kanishk Gandhi, Denise Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah D Goodman. Stream of search (SoS): Learning to search in language. *arXiv [cs.LG]*, 2024. URL http://arxiv.org/abs/2404.03683.

Kanishk Gandhi, Ayush Chakravarthy, Anikait Singh, Nathan Lile, and Noah D Goodman. Cognitive behaviors that enable self-improving reasoners, or, four habits of highly effective STaRs. *arXiv [cs.CL]*, 2025. URL http://arxiv.org/abs/2503.01307.

Alex Havrilla, Sharath Raparthy, Christoforus Nalmpantis, Jane Dwivedi-Yu, Maksym Zhuravinskyi, Eric Hambro, and Roberta Raileanu. GLoRe: When, where, and how to improve LLM reasoning via global and local refinements. *arXiv [cs.CL]*, 2024. URL http://arxiv.org/abs/2402.10963.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. *arXiv [cs.LG]*, 2021. URL http://arxiv.org/abs/2103.03874.

Zhenyu Hou, Xin Lv, Rui Lu, Jiajie Zhang, Yujiang Li, Zijun Yao, Juanzi Li, Jie Tang, and Yuxiao Dong. Advancing language model reasoning through reinforcement learning and inference scaling. *arXiv [cs.LG]*, 2025. URL http://arxiv.org/abs/2501.11651.

Robert Irvine, Douglas Boubert, Vyas Raina, Adian Liusie, Ziyi Zhu, Vineet Mudupalli, Aliaksei Korshuk, Zongyi Liu, Fritz Cremer, Valentin Assassi, Christie-Carol Beauchamp, Xiaoding Lu, Thomas Rialan, and William Beauchamp. Rewarding chatbots for real-world engagement with millions of users. *arXiv [cs.CL]*, 2023. URL http://arxiv.org/abs/2303.06135.

Amirhossein Kazemnejad, Milad Aghajohari, Eva Portelance, Alessandro Sordoni, Siva Reddy, Aaron Courville, and Nicolas Le Roux. VinePPO: Unlocking RL potential for LLM reasoning through refined credit assignment. *arXiv [cs.LG]*, 2024. URL http://arxiv.org/abs/2410.01679.

Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, Lei M Zhang, Kay McKinney, Disha Shrivastava, Cosmin Paduraru, George Tucker, Doina Precup, Feryal Behbahani, and Aleksandra Faust. Training language models to self-correct via reinforcement learning. *arXiv [cs.LG]*, 2024. URL http://arxiv.org/abs/2409.12917.

Ariel N Lee, Cole J Hunter, and Nataniel Ruiz. Platypus: Quick, cheap, and powerful refinement of LLMs. *arXiv [cs.CL]*, 2023. URL http://arxiv.org/abs/2308.07317.

Kuang-Huei Lee, Ian Fischer, Yueh-Hua Wu, Dave Marwood, Shumeet Baluja, Dale Schuurmans, and Xinyun Chen. Evolving deeper LLM thinking. *arXiv [cs.AI]*, 2025. URL http://arxiv.org/abs/2501.09891.

Noam Levi. A simple model of inference scaling laws. *arXiv [stat.ML]*, 2024. URL http://arxiv.org/abs/2410.16377.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv [cs.LG]*, 2023. URL http://arxiv.org/abs/2305.20050.

Jiacheng Liu, Andrew Cohen, Ramakanth Pasunuru, Yejin Choi, Hannaneh Hajishirzi, and Asli Celikyilmaz. Don't throw away your value model! generating more preferable text with value-guided monte-carlo tree search decoding. *arXiv [cs.CL]*, 2023. URL http://arxiv.org/abs/2309.15028.

Jieyi Long. Large language model guided tree-of-thought. *arXiv [cs.AI]*, 2023. URL http://arxiv.org/abs/2305.08291.

Liangchen Luo, Yinxiao Liu, Rosanne Liu, Samrat Phatale, Meiqi Guo, Harsh Lara, Yunxuan Li, Lei Shu, Yun Zhu, Lei Meng, Jiao Sun, and Abhinav Rastogi. Improve mathematical reasoning in language models by automated process supervision. *arXiv [cs.CL]*, 2024. URL http://arxiv.org/abs/2406.06592.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling. *arXiv [cs.CL]*, 2025. URL http://arxiv.org/abs/2501.19393.

Aleksei Maslakov And Papadimas. Sudoku solver with step-by-step guidance, 2023. URL https://github.com/unmade/dokusan,2023.

Debjit Paul, Mete Ismayilzada, Maxime Peyrard, Beatriz Borges, Antoine Bosselut, Robert West, and Boi Faltings. REFINER: Reasoning feedback on intermediate representations. *arXiv [cs.CL]*, 2023. URL http://arxiv.org/abs/2304.01904.

Yuxiao Qu, Tianjun Zhang, Naman Garg, and Aviral Kumar. Recursive IntroSpEction: Teaching language model agents how to self-improve. *arXiv [cs.LG]*, 2024. URL http://arxiv.org/abs/2407.18219.

David Radcliffe. 3 million sudoku puzzles with ratings. Website, 2020. URL https://www.kaggle.com/datasets/radcliffe/3-million-sudoku-puzzles-with-ratings.

William Saunders, Catherine Yeh, Jeff Wu, Steven Bills, Long Ouyang, Jonathan Ward, and Jan Leike. Self-critiquing models for assisting human evaluators. *arXiv [cs.CL]*, 2022. URL http://arxiv.org/abs/2206.05802.

Kulin Shah, Nishanth Dikkala, Xin Wang, and Rina Panigrahy. Causal language modeling can elicit search and reasoning capabilities on logic puzzles. *arXiv [cs.LG]*, 2024. URL http://arxiv.org/abs/2409.10502.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y K Li, Y Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv [cs.CL]*, 2024. URL http://arxiv.org/abs/2402.03300.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A flexible and efficient RLHF framework. *arXiv preprint arXiv: 2409. 19256*, 2024.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *arXiv [cs.CL]*, 2021. URL http://arxiv.org/abs/2104.09864.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-shepherd: Verify and reinforce LLMs step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9426–9439, Stroudsburg, PA, USA, 2024. Association for Computational Linguistics. URL http://dx.doi.org/10.18653/v1/2024.acl-long.510.

Tianlu Wang, Ping Yu, Xiaoqing Ellen Tan, Sean O'Brien, Ramakanth Pasunuru, Jane Dwivedi-Yu, Olga Golovneva, Luke Zettlemoyer, Maryam Fazel-Zarandi, and Asli Celikyilmaz. Shepherd: A critic for language model generation. *arXiv [cs.CL]*, 2023. URL http://arxiv.org/abs/2308.04592.

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. Generating sequences by learning to self-correct. *arXiv [cs.CL]*, 2022. URL http://arxiv.org/abs/2211.00053.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. *arXiv [cs.AI]*, 2024. URL http://arxiv.org/abs/2408.00724.

Yuxi Xie, Kenji Kawaguchi, Yiran Zhao, Xu Zhao, Min-Yen Kan, Junxian He, and Qizhe Xie. Self-evaluation guided beam search for reasoning. *arXiv [cs.CL]*, 2023. URL http://arxiv.org/abs/2305.00633.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. DeepSeek-prover: Advancing theorem proving in LLMs through large-scale synthetic data. *arXiv [cs.AI]*, 2024. URL http://arxiv.org/abs/2405.14333.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412. 15115*, 2024.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *arXiv [cs.CL]*, 2023a. URL http://arxiv.org/abs/2305.10601.

Weiran Yao, Shelby Heinecke, Juan Carlos Niebles, Zhiwei Liu, Yihao Feng, Le Xue, Rithesh Murthy, Zeyuan Chen, Jianguo Zhang, Devansh Arpit, Ran Xu, Phil Mui, Huan Wang, Caiming Xiong, and Silvio Savarese. Retroformer: Retrospective large language agents with policy gradient optimization. *arXiv [cs.CL]*, 2023b. URL http://arxiv.org/abs/2308.02151.

Seonghyeon Ye, Yongrae Jo, Doyoung Kim, Sungdong Kim, Hyeonbin Hwang, and Minjoon Seo. SelFee: Iterative self-revising LLM empowered by self-feedback generation. Blog post, 2023. URL https://kaistai.github.io/SelFee/.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D Goodman. STaR: Bootstrapping reasoning with reasoning. *arXiv [cs.LG]*, 2022. URL http://arxiv.org/abs/2203.14465.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. Planning with large language models for code generation. *arXiv [cs.LG]*, 2023. URL http://arxiv.org/abs/2303.05510.

Yunxiang Zhang, Muhammad Khalifa, Lajanugen Logeswaran, Jaekyeom Kim, Moontae Lee, Honglak Lee, and Lu Wang. Small language models need strong verifiers to self-correct reasoning. *arXiv [cs.CL]*, 2024. URL http://arxiv.org/abs/2404.17140.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. *arXiv [cs.AI]*, 2023. URL http://arxiv.org/abs/2310.04406.

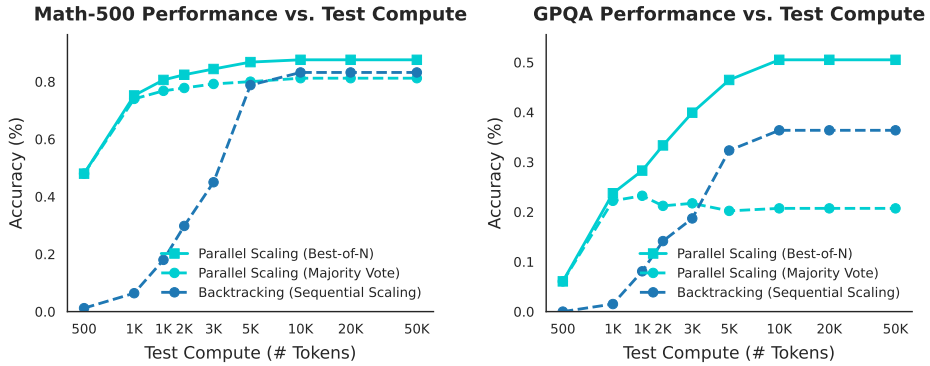


Figure 7: **Evaluating backtracking on real LLMs.** *Left:* On MATH-500, we compare the S1 model (fine-tuned on backtracking traces) using sequential decoding with budget forcing, against its base model (Qwen2.5-32B-Instruct) using parallel sampling. The backtracking model *underperforms* at low compute but narrows the gap at higher budgets. *Right:* On GPQA, the same backtracking setup *outperforms* parallel sampling in a multiple-choice reasoning setting. **This comparison generalizes our conclusion from synthetic settings to real LLMs.**

# A Backtracking Analysis on Math Reasoning with LLMs

## A.1 Experimental Setup

To complement our synthetic experiments, we conduct an evaluation on real-world math problems to examine whether **backtracking remains effective under equal test-time compute**. We compare two approaches:

- **Backtracking model**: fine-tuned on solution traces that include explicit self-correction and step-by-step reflection.
- **Direct solution model**: the base model without backtracking fine-tuning, using parallel sampling (with majority voting for final correct answer) at inference.

To control test-time compute, we use the **budget forcing** technique introduced in (Muennighoff et al., 2025). This enables a fair comparison across models with differing reasoning styles.

**Backtracking Model.** We adopt the `S1` checkpoint from Muennighoff et al. (2025), a model trained on solution traces distilled from DeepSeekR1. These traces exhibit explicit backtracking behaviors—identifying and correcting earlier mistakes. We generate outputs with temperature $T = 0.7$ under budget forcing and evaluate on the `MATH-500`.

**Direct Solution Model.** For fair comparison, we use the same base model as `S1`—Qwen2.5-32B-Instruct—without backtracking fine-tuning. We sample $N = 1$ to 8 completions with temperature $T = 0.7$, and report both **Best-of-$N$** and **Majority Vote** accuracy.

### Results and Interpretation

Figure 7 (*left*) presents accuracy under matched compute budgets. We observe that at **low compute budgets** the backtracking model underperforms due to its verbose reasoning traces. At **higher budgets**, backtracking matches and slightly exceeds the performance of parallel sampling. This mirrors trends observed in the **CountDown** (Section 4.3), and suggests that while backtracking introduces overhead, it yields benefits when sufficient compute is available.

To form a sharp contrast, we reproduce results from (Muennighoff et al., 2025) on `GPQA-Diamond` (Figure 7, *right*), which shows that the same backtracking model significantly outperforms parallel sampling—even at lower budgets—in a multiple-choice setting. This contrast highlights that **the effectiveness of backtracking is task-dependent**.

This real-world evaluation supports our synthetic findings: **backtracking improves performance under compute constraints**, but its advantage depends on the task structure. On open-ended math problems, the benefit is most pronounced at higher budgets. On structured tasks like multiple-choice QA, gains can appear even earlier. Overall, our conclusions **generalize beyond synthetic settings**.

# B Related Work Extended

## B.1 Test-time computation scaling

A growing body of work has explored how to improve language model performance by scaling test-time computation. These approaches typically fall into two broad categories: **parallel** and **sequential** search. Parallel methods sample multiple solutions independently and select the best one using predefined criteria—such as majority voting or external reward models—as seen in Best-of-$N$ techniques (Brown et al., 2024; Irvine et al., 2023; Levi, 2024). These methods often rely on outcome-based reward models that score complete solutions (Xin et al., 2024; Ankner et al., 2024).

In contrast, sequential methods iteratively refine reasoning by conditioning on previous attempts. This class includes stepwise improvement methods(Ankner et al., 2024; Hou et al., 2025; Lee et al., 2025), where each new trajectory builds on earlier outputs, enabling the model to adapt its reasoning dynamically. Other research works have also explored using the search process itself to improve model reasoning capabilities, either during inference or by integrating the feedback into training(Wang et al., 2024; Luo et al., 2024). While these methods can reduce redundancy, they typically require more compute per sample and may suffer from compounding errors.

Tree-based approaches, such as Monte Carlo Tree Search (MCTS) and guided beam search, represent a hybrid between parallel and sequential strategies(Gandhi et al., 2024; Liu et al., 2023; Zhang et al., 2023; Zhou et al., 2023; Choi et al., 2023; Xie et al., 2023). These methods often leverage process reward models, which assign value to intermediate reasoning steps rather than full outputs(Lightman et al., 2023; Wang et al., 2024; Wu et al., 2024). REBASE(Wu et al., 2024), for example, uses a process reward model to guide exploration and pruning in tree search, and has been shown to outperform both sampling-based methods and traditional MCTS.

## B.2 Self-correction and backtracking

Search and backtracking are inherently tied to self-correction, as they enable models to revisit earlier decisions and recover from errors—a critical capability for multi-step reasoning. Teaching language models to self-correct has been approached through fine-tuning on revision demonstrations from humans or stronger models(Saunders et al., 2022; Ye et al., 2023; Qu et al., 2024), as well as through synthetic data generation and handcrafted augmentation(Paul et al., 2023; Wang et al., 2023; Lee et al., 2023). Reward-based methods provide another avenue, using outcome- or process-level signals to differentiate good and bad reasoning trajectories, often framed as implicit policy learning(Welleck et al., 2022; Akyürek et al., 2023; Zhang et al., 2024). Some methods further incorporate search, critique generation, or separate correction modules to enhance reasoning quality(Yao et al., 2023b; Havrilla et al., 2024). In contrast, using two structured games, we investigate the tradeoffs of teaching models to backtrack via search traces versus allowing them to learn purely from correct solutions.

## B.3 Reinforcement learning for LLM reasoning

Reinforcement learning (RL) has emerged as a powerful framework for improving the reasoning abilities of language models. While early work applied off-policy and on-policy RL methods to guide models toward verifiable outcomes(Zelikman et al., 2022; Kazemnejad et al., 2024), recent approaches have shown that even simplified algorithms like GRPO can lead to significant performance gains and the emergence of in-context search behavior (DeepSeek-AI et al., 2025; Shao et al., 2024; DeepSeek-AI et al., 2025). These advances suggest that RL can help models autonomously discover more effective reasoning strategies, even without explicit reward models or structured search. However, not all models benefit

equally from RL, and it remains unclear what properties make a model amenable to learning through reinforcement. Our work contributes to this question by examining how backtracking models, when trained with GRPO, can discover novel solution strategies—while no-backtracking models show limited or mixed gains.

# C Experiment details

## C.1 Additional details on game, data generation

**CountDown tree size computation.** CountDown has an exponentially growing search space with respect to the number of candidate numbers. If the current state has $N$ available numbers, there are $\binom{N}{2} \times 4$ possible actions (selecting a pair and one of four operations), and the depth of the tree is $N - 1$. For games with four candidate numbers, the complete search tree contains 1,152 nodes.

**CountDown search.** To generate DFS search data, we use a sum heuristic to guide the search order and prune nodes. This heuristic measures the distance between the sum of all input numbers and the target number, and prunes a node if the absolute distance exceeds the target. This approach is inspired by Gandhi et al. (2024), who also consider an alternative—the multiply heuristic—which measures the minimum distance between the input set and the factors of the target. However, in our experiments, both heuristics yield similar performance: for a fixed number of visited nodes, DFS with either heuristic solves approximately the same number of games.

**Sudoku rule.** In a Sudoku game, the player is given a $9 \times 9$ grid in which each cell must be filled with a digit from 1 to 9. The puzzle is subject to three constraints: each row, each column, and each of the nine $3 \times 3$ subgrids must contain all digits from 1 to 9 exactly once. Given a partially filled grid, the objective is to fill in the remaining cells such that all constraints are satisfied.

**Sudoku data and tokenization.** To represent the Sudoku board for language models, we encode each cell as a position-value pair: $(x, y) = v$, where $(x, y)$ denotes the grid location and $v$ is the cell's value. The model receives the initial board as a list of known $(x, y) = v$ pairs and generates the solution by predicting the values for the remaining cells. We generate **backtracking traces** by serializing the full DFS traversal. For the **direct solution model**, we prune each trace to include only the final solution path.

**Scoring.** For CountDown, a solution is correct only if it adheres to game rules and correctly achieves the target number. For Sudoku, correctness requires fully solving the board, with no partial credit given for incomplete but correct boards. Models are tested on 200 unseen problems per game. The same scoring function is used as the reward function in GRPO (Section 6)

## C.2 Additional details on model architecture

Model hyperparameters can be found in Table 1.

| Model Size | Hidden Size | Layers | Attn Heads | Intermediate Size | KV Heads |
|---|---|---|---|---|---|
| 3M | 256 | 6 | 4 | 512 | 1 |
| 17M | 512 | 8 | 4 | 1024 | 1 |
| 38M | 512 | 10 | 8 | 2048 | 2 |
| 144M | 1024 | 12 | 8 | 3072 | 2 |

Table 1: Qwen2.5-style architecture configurations for the four model sizes used in our experiments.

## C.3 Training hyperparameter

Training hyperparameters can be found in Table 2. We train all models on 2 NVIDIA H100 80GB HBM3 GPUs.

| Hyperparameter | Value |
|---|---|
| *Optimization* | |
| Learning rate | $1 \times 10^{-5}$ |
| Weight decay | 0.01 |
| *Learning Rate Schedule* | |
| Scheduler type | Cosine |
| Warmup steps | 1 |
| *Training Setup* | |
| Epochs | 30 |
| Batch size (backtracking model) | 32 |
| Batch size (direct solution model) | 64 |
| Context length (backtracking model) | 4096 |
| Context length (direct solution model) | 512 |
| *Tokenizer* | |
| Tokenizer size (CountDown) | 74 |
| Tokenizer size (Sudoku) | 110 |

Table 2: Training hyperparameters used for all experiments. Batch size and context length vary based on model type.

## D FLOP computation

To compare backtracking and direct solution models under a fixed compute budget, we estimate inference FLOPs based on model architecture and generation length $T$. We use a simplified transformer FLOP computation that accounts for per-token operations across all layers.

Below is a list of architectural and generation parameters:

- $d_{\text{model}}$: hidden dimension
- $d_{\text{kv}}$: key/value dimension [2]
- $d_{\text{ff}}$: intermediate (feedforward) dimension
- $L$: number of layers
- $T$: number of generated tokens (i.e., context length)
- $N$: number of sequences generated (e.g., in best-of-$N$ sampling)

### D.1 Step-by-step FLOPs Calculation

**1. Per-layer linear FLOPs per token.** We break down the linear FLOPs for each transformer layer into attention and MLP components:

- **Self-attention:**
  - Query projection: $d_{\text{model}} \times d_{\text{model}}$
  - Key projection: $d_{\text{model}} \times d_{\text{kv}}$
  - Value projection: $d_{\text{model}} \times d_{\text{kv}}$
  - Output projection: $d_{\text{model}} \times d_{\text{model}}$

  This results in a total of:

$$\text{FLOPs}_{\text{attention-linear}} = 2d_{\text{model}}^2 + 2d_{\text{model}}d_{\text{kv}}$$

- **MLP (Feedforward):**
  MLP layers include following components:
  - Gate projection
  - Up projection
  - Down projection

  Each of these MLP layers costs: $d_{\text{model}} \times d_{\text{ff}}$, giving:

$$\text{FLOPs}_{\text{mlp}} = 3d_{\text{model}}d_{\text{ff}}$$

[2] key/value dimension is different from hidden dimension because of GQA (Ainslie et al., 2023)

Combining both components, the total per-token linear cost per layer is:

$$\text{FLOPs}_{\text{linear}} = 2d_{\text{model}}^2 + 2d_{\text{model}}d_{\text{kv}} + 3d_{\text{model}}d_{\text{ff}}$$

**2. Quadratic attention cost.** Self-attention involves computing interactions between all token pairs, resulting in a quadratic cost:

$$\text{FLOPs}_{\text{attention}} = d_{\text{model}} \cdot \frac{T(T+1)}{2}$$

**3. Total generation cost per sequence.** Each token attends to all previous tokens across all $L$ layers. The generation cost for a single sequence is:

$$\text{FLOPs}_{\text{gen}} = L \cdot (\text{FLOPs}_{\text{linear}} \cdot T + \text{FLOPs}_{\text{attention}})$$

**4. Total inference FLOPs.** For $N$ sequences (e.g., best-of-$N$ sampling), the total inference cost is:

$$\text{FLOPs}_{\text{total}} = N \cdot \text{FLOPs}_{\text{gen}}$$

We do not include auxiliary operations such as token embedding and softmax, weight norm, as their contribution is negligible compared to the transformer layers. All FLOPs reported in our experiments use this formula, with model configurations listed in Table 1.

## E Majority voting versus best-of-n

In this work, we primarily use the best-of-$n$ metric to evaluate the direct solution model. This metric is suitable for tasks where verifying the correctness of a solution is trivial, whereas solving the task itself is challenging. Many real-world problems, such as coding tasks and combinatorial optimization, fall into this category. Conversely, for problems where verification is difficult, metrics such as majority voting may be more appropriate.

To illustrate this point, we additionally evaluate the CountDown direct solution model using both metrics in Figure 8. For majority voting, we generate n solutions per test problem, select the most frequently occurring solution (breaking ties randomly), and evaluate its correctness.

We find that the majority-voting performance closely approximates the direct solution model's one-shot accuracy (i.e., best-of-$n$ with $n$=1). However, majority voting is less suitable for our task for several reasons. First, the CountDown game frequently has multiple correct solutions, so selecting the majority solution path can fail to detect cases where the model generates different but equally valid solutions. Second, while majority voting is appropriate in real-world LLM scenarios—such as mathematical reasoning—where distinct solution paths converge to the same final boxed answer, in our synthetic setting, where models are trained from scratch, majority voting essentially becomes a noisy proxy for greedy decoding (sampling at temperature $T = 0$). Thus, we expect and observe majority voting accuracy to closely track pass@1 accuracy.

In summary, given the characteristics of our task and the controlled experimental setup, best-of-$n$ remains a valid and preferred metric for evaluating direct solution models.

## F Dependence on depth of the search tree

### F.1 Search tree depth

Why do backtracking models perform well on Sudoku but underperform on CountDown, even when both are trained on DFS search traces? We argue that task characteristics—particularly those beyond our control in real—world settings—play a key role in determining whether backtracking is test-time-compute-efficient. A major difference between the two games lies in the depth of their search trees (Figure 2). In hard Sudoku puzzles, only 20 out of 81 cells are pre-filled, leaving 50–60 cells to solve. This results in deep search trees with

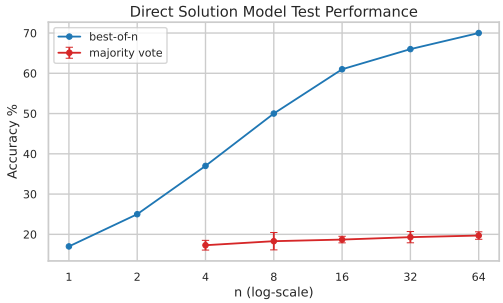


Figure 8: **Majority voting versus best-of-$n$ for CountDown direct solution model.** For CountDown, verification is much easier than solving the problem. Therefore, best-of-$n$ as a performance is justified. Additionally, we also examine majority voting performance. However, CountDown solutions are not unique, majority voting is not the most suitable way to measure model performances.

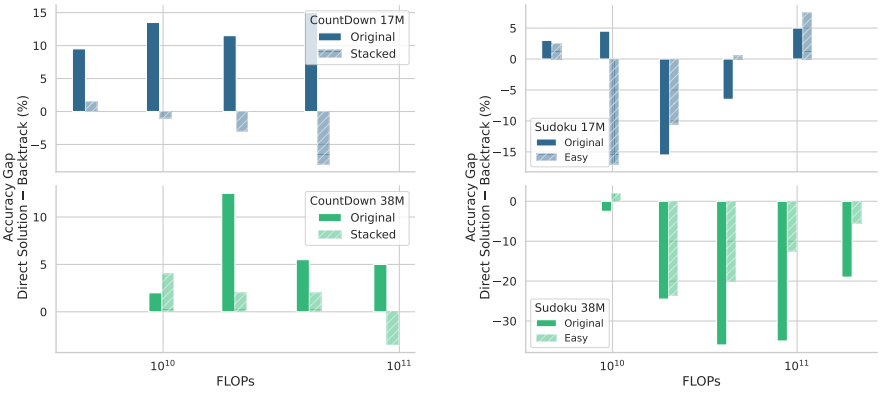


Figure 9: **The efficacy of backtracking depends on search tree depth.** *Left:* We introduce a variation of the CountDown game—stacked CountDown—to **increase** the search tree depth. In the original CountDown game (*solid bars*), the direct solution model consistently **outperforms** the backtracking model, shown by a positive performance gap. In the stacked version (*slanted bars*), this gap is significantly reduced or even reversed, indicating that backtracking becomes more compute-efficient at greater depths. *Right:* We introduce a variation of Sudoku—easy Sudoku—where the initial board has more pre-filled cells to **decrease** the search tree depth. In the original setting, the 38M direct solution model (*bottom, solid bars*) **underperforms** the backtracking model. In the shallow Sudoku variant (*slanted bars*), the performance gap narrows across compute budgets. For the 17M models (*top*), the results are less conclusive.

extensive trial-and-error, with many backtracking steps. In contrast, CountDown (in our setup) uses 4 candidate numbers, limiting the search tree depth to just 3. We hypothesize that backtracking models excels at tasks with deeper search trees, while shallow trees make parallel strategies (i.e., direct solution model) more effective. To test this, we design a variant of CountDown with increased search depth and a variant of Sudoku with reduced depth.

### F.2 A deeper CountDown

**Set up** To increase the search tree depth in CountDown, one might naively scale up the number of candidate numbers. However, this approach quickly leads to exponential growth in tree width: with 4 candidates, the tree contains 1,152 nodes; with 5 candidates, it grows to 46,080. To prevent the exponential growth in the number of search paths, we design a stacked CountDown variant that increases depth while controlling tree width. In this setup, the player is given 8 candidate numbers and a final target. The first 4 numbers must be used

to reach the 5th number ("a partial goal"), and the remaining 4 numbers must then be used to reach the final target. This effectively stacks two CountDown problems, increasing depth without combinatorial explosion. We generate training data for both backtracking and no-backtracking models following the same procedure as in Section 3.1.2, with examples provided in Appendix H (Figure 16). We train a 17M as well as a 38M model until validation loss has converged, and test on 200 unseen problems.

**Results** In Figure 9 (*left*), we compare the performance gap between the direct solution model and the backtracking model, measured by the difference in test accuracy. In the original CountDown setting (*solid bars*), the direct solution model consistently **outperforms** the backtracking model across all test compute budgets. However, in the stacked CountDown variant (*slanted bars*), the performance gap narrows significantly—and in some cases, reverses. The sign reverse indicates the backtracking model now **outperforms** the direct solution model. These results support our hypothesis: in CountDown, backtracking becomes more compute-efficient as the search tree depth increases. We observe this trend across both 17M and 38M models.

### F.3 A shallower Sudoku

**Set up** To reduce the search tree depth in Sudoku, we generate easier boards by increasing the number of initially filled cells. Specifically, we take the original 3M Sudoku dataset Radcliffe (2020) and apply the direct solution model (Section 3.2.2) to correctly fill 10 additional cells. This increases the average number of pre-filled cells from 20 to around 30, effectively decreasing search tree depth. We generate both backtracking and direct solution training data following the same procedure in Section 3.2.2. Models with 17M and 38M parameters are trained to convergence and evaluated on 200 unseen problems.

**Results** In Figure 9 (*right*), we show the performance gap between the direct solution and backtracking models, measured by the difference in test accuracy. In the original (hard) Sudoku setting, the 38M direct solution model consistently **underperforms** the backtracking model, as indicated by the negative gaps (*solid green bars*). In the shallow-Sudoku variant (*slanted bars*), these gaps are reduced across all test-time compute budgets for the 38M model. The trend is less clear for the 17M model, where the performance difference remains small in both settings. Overall, these results support our hypothesis: in Sudoku, backtracking becomes more test-time-compute-efficient when the search tree is deeper.

## G Additional results

### G.1 Exploring different CountDown strategies

We analyze different search strategies for CountDown, including DFS and BFS with varying beam widths. For each strategy, we tokenize the resulting backtracking trace and measure number of tokens used in each search trace. The goal is to identify which strategy that finds correct solutions with the fewest tokens (Figure 10). The results show no clear winner. BFS with a smaller beam width produces shorter traces by exploring fewer nodes, but this comes at the cost of missing correct solutions more frequently. Increasing the beam width improves solution coverage but leads to longer traces due to broader exploration.

In contrast, DFS produces more uniformly distributed trace lengths but suffers from a specific failure mode: it may prune the correct path early and terminate prematurely. These failures appear as short but incorrect traces, visible as the left-most orange bars in Figure 10 (*bottom*).

### G.2 Compare think-backtrack and backtrack

Table 3 further shows a confusion matrix comparing the original and think-backtrack models. The backtracking model solves 102 test problems in total with maximum test-time compute budget (4096 tokens). Out of those 102 problems, the think-backtrack model solves most of them. This evidence further shows that by training on shortened search traces, the model learns to internalize parts of its thinking without sacrificing performances.

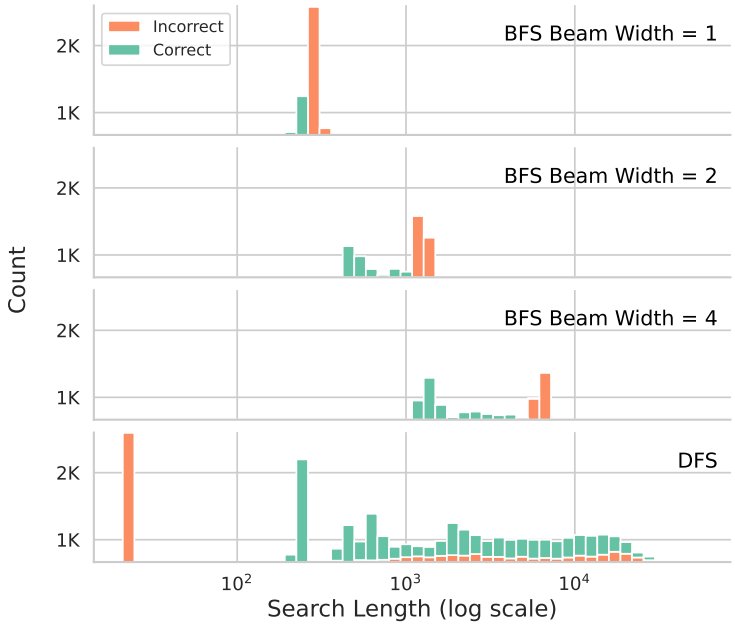


Figure 10: **Examine different search strategies for CountDown.** Beyond DFS, we experiment with Bread-First-Search (BFS) with different beam widths. We tokenize the search trace and measure the number of tokens as search length. There is not one search algorithm that is optimal to generate both short and correct solution traces.

| | **T-B Correct** | **T-B Incorrect** |
|---|---|---|
| **B Correct** | 83 | 19 |
| **B Incorrect** | 41 | 57 |

Table 3: Confusion matrix between Think-Backtrack (T-B) and Backtrack (B) models.

### G.3 Supervised learning training curve

During training, we set the maximum epochs to 30 epochs and allow early stopping. All models converge before 30 epochs and we early stop training when the validation loss has converged on log-log scale. Figure 11, 12 show the training curve for both models and for CountDown and Sudoku.

### G.4 Additional GRPO plots

In Figure 3 (Section 4.2.1), we used the number of mistakes as a proxy for comparing search strategies. To further demonstrate that the backtracking model fine-tuned with GRPO discovers new strategies, we repeat the same analysis in Figure 13 (*right*). Compared to the original backtracking model (Figure 13, *left*), the post-GRPO model solves many problems with a different number of mistakes than the number of mistakes made by DFS. This shift indicates that the model is no longer tightly aligned with the original search trace and has discovered alternative, more diverse solution paths. Figure 6 (*left*) quantifies the above qualitative observation.

## H Data sample

Figure 14 shows an example of a CountDown game and the training data. Figure 15 shows an example of a Sudoku game and the training data. Figure 16 shows an example of stacked-CountDown variation and the training data.

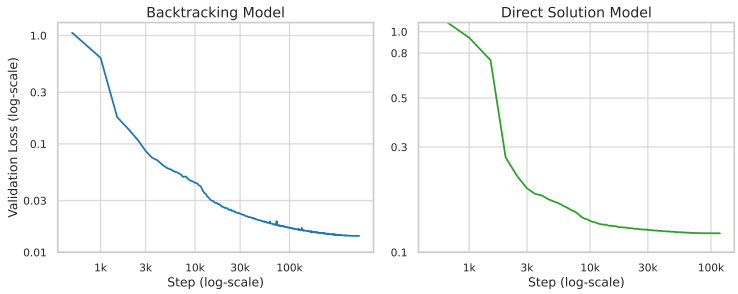


Figure 11: **CountDown validation loss.** *Left:* Backtracking model. *Right:* Direct solution model.

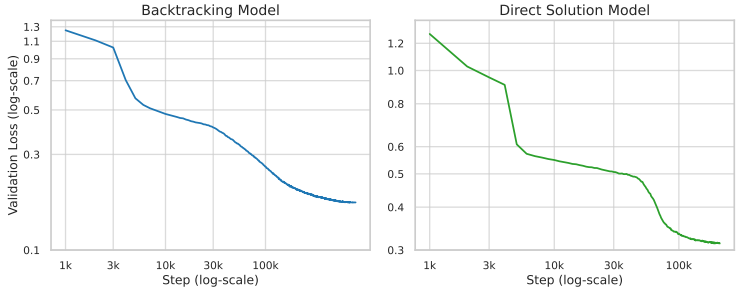


Figure 12: **Sudoku validation loss.** *Left:* Backtracking model. *Right:* Direct solution model.

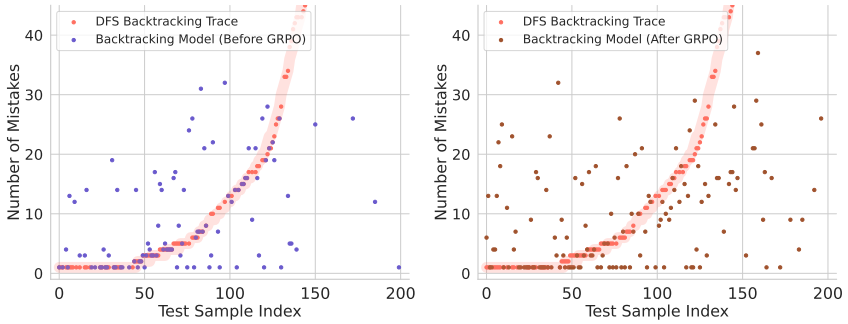


Figure 13: **Backtracking model can discover some new search strategies.** Post GRPO, the backtracking model discover new strategies: In the *right* panel, for each problem, the post-GRPO model makes a different number of mistakes compared to DFS trace.

**Example of a CountDown game**

```
Target: 16      Candidates: [96, 11, 78, 22]
```

**DFS Search Trace**

```
Current State: 16:[96, 11, 78, 22], Operations: []
Exploring Operation: 96-78=18, Resulting Numbers: [11, 22, 18]
Generated Node #0,0: 16:[11, 22, 18] Operation: 96-78=18
Moving to Node #0,0
Current State: 16:[11, 22, 18], Operations: ['96-78=18']
Exploring Operation: 22-11=11, Resulting Numbers: [18, 11]
Generated Node #0,0,0: 16:[18, 11] Operation: 22-11=11
Moving to Node #0,0,0
Current State: 16:[18, 11], Operations: ['96-78=18', '22-11=11']
Exploring Operation: 18-11=7, Resulting Numbers: [7]
7,16 unequal: No Solution
Moving to Node #0,0,0
Current State: 16:[18, 11], Operations: ['96-78=18', '22-11=11']
Exploring Operation: 18+11=29, Resulting Numbers: [29]
29,16 unequal: No Solution
Moving to Node #0,0
Current State: 16:[11, 22, 18], Operations: ['96-78=18']
Exploring Operation: 18-11=7, Resulting Numbers: [22, 7]
Generated Node #0,0,1: 16:[22, 7] Operation: 18-11=7
Moving to Node #0,0,1
Current State: 16:[22, 7], Operations: ['96-78=18', '18-11=7']
Exploring Operation: 22-7=15, Resulting Numbers: [15]
15,16 unequal: No Solution
Moving to Node #0,0,1
Current State: 16:[22, 7], Operations: ['96-78=18', '18-11=7']
Exploring Operation: 22+7=29, Resulting Numbers: [29]
29,16 unequal: No Solution
Moving to Node #0,0
Current State: 16:[11, 22, 18], Operations: ['96-78=18']
Exploring Operation: 22/11=2, Resulting Numbers: [18, 2]
Generated Node #0,0,2: 16:[18, 2] Operation: 22/11=2
Moving to Node #0,0,2
Current State: 16:[18, 2], Operations: ['96-78=18', '22/11=2']
Exploring Operation: 18-2=16, Resulting Numbers: [16]
16,16 equal: Goal Reached
```

**Direct Solution Trace**

```
Current State: 16:[96, 11, 78, 22], Operations: []
Exploring Operation: 96-78=18, Resulting Numbers: [11, 22, 18]
Generated Node #2: [11, 22, 18] from Operation: 96-78=18
Current State: 16:[11, 22, 18], Operations: ['96-78=18']
Exploring Operation: 22/11=2, Resulting Numbers: [18, 2]
Generated Node #3: [18, 2] from Operation: 22/11=2
Current State: 16:[18, 2], Operations: ['96-78=18', '22/11=2']
Exploring Operation: 18-2=16, Resulting Numbers: [16]
16,16 equal: Goal Reached
```

Figure 14: **Example of training data for CoutnDown.** *Top*: game setup. *Middle*: full DFS search trace for training backtracking model. *Bottom*: correct solution path for training direct solution model.

Initial Board

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | | 2 | 4 | | | | | 5 |
| | | | 1 | | | | 7 | |
| | 8 | 3 | | | | | 9 | |
| | | | | 5 | | | 1 | |
| 2 | | 9 | | | | | | 7 |
| | | | 3 | | 2 | | | |
| | | 8 | | 4 | | | | |
| | 4 | 7 | | 8 | | | | 9 |
| 6 | | | 2 | 1 | | | | 3 |

Solved Board

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | 1 | 2 | 4 | 9 | 8 | 6 | 3 | 5 |
| 9 | 6 | 4 | 1 | 3 | 5 | 2 | 7 | 8 |
| 5 | 8 | 3 | 7 | 2 | 6 | 1 | 9 | 4 |
| 8 | 7 | 6 | 9 | 5 | 4 | 3 | 1 | 2 |
| 2 | 3 | 9 | 8 | 6 | 1 | 4 | 5 | 7 |
| 4 | 5 | 1 | 3 | 7 | 2 | 9 | 8 | 6 |
| 3 | 2 | 8 | 5 | 4 | 9 | 7 | 6 | 1 |
| 1 | 4 | 7 | 6 | 8 | 3 | 5 | 2 | 9 |
| 6 | 9 | 5 | 2 | 1 | 7 | 8 | 4 | 3 |

**Example of a Sudoku game**

START (0, 0) = 7 (0, 2) = 2 (0, 3) = 4 (0, 8) = 5 (1, 3) = 1 (1, 7) = 7 (2, 1) = 8 (2, 2) = 3 (2, 7) = 9 (3, 4) = 5 (3, 7) = 1 (4, 0) = 2 (4, 2) = 9 (4, 8) = 7 (5, 3) = 3 (5, 5) = 2 (6, 2) = 8 (6, 4) = 4 (7, 1) = 4 (7, 2) = 7 (7, 4) = 8 (7, 8) = 9 (8, 0) = 6 (8, 3) = 2 (8, 4) = 1 (8, 8) = 3 solving

**DFS Search Trace**

SOL_START (4, 4) = 6 (8, 2) = 5 (4, 3) = 8 (8, 1) = 9 (8, 5) = 7 GUESS: (0, 1) [1, 6] = 1 (0, 1) = 1 GUESS: (0, 4) [3, 9] = 3 (0, 4) = 3 GUESS: (0, 6) [6, 8] = 6 (0, 6) = 6 (0, 7) = 8 (8, 7) = 4 (0, 5) = 9 (1, 4) = 2 (1, 8) = 4 (2, 4) = 7 (3, 5) = 4 (4, 5) = 1 (5, 4) = 9 (8, 6) = 8 (1, 2) = 6 (1, 6) = 3 (3, 3) = 7 (1, 1) = 5 (1, 5) = 8 (2, 0) = 4 (4, 1) = 3 (4, 7) = 5 (5, 6) = 4 (5, 7) = 6 (5, 8) = 8 (6, 1) = 2 (7, 7) = 2 (1, 0) = 9 (3, 0) = 8 (3, 1) = 6 (3, 8) = 2 (5, 1) = 7 (5, 2) = 1 (2, 8) = 1 (3, 6) = 9 (5, 0) = 5 (6, 8) = 6 (2, 6) = 2 NO_CANDIDATE: (3, 2) NO CANDIDATE: (0, 6) REVERT: (0, 6) [6, 8] = NONE GUESS: (0, 6) [6, 8] = 8 (0, 6) = 8 (0, 7) = 6 (8, 6) = 4 (8, 7) = 8 (0, 5) = 9 (1, 4) = 2 (1, 6) = 3 (1, 8) = 4 (2, 4) = 7 (3, 5) = 4 (4, 5) = 1 (4, 6) = 5 (5, 4) = 9 (5, 6) = 6 (5, 7) = 4 (5, 8) = 8 (1, 2) = 6 (3, 3) = 7 (3, 8) = 2 (4, 1) = 3 (5, 2) = 1 (6, 1) = 2 (6, 7) = 5 (7, 7) = 2 (1, 1) = 5 (1, 5) = 8 (2, 0) = 4 (2, 8) = 1 (3, 0) = 8 (3, 1) = 6 (3, 6) = 9 (5, 0) = 5 (5, 1) = 7 (6, 8) = 6 (7, 6) = 1 (1, 0) = 9 (2, 6) = 2 (6, 3) = 9 (6, 5) = 3 (6, 6) = 7 (7, 0) = 3 (6, 0) = 1 NO_CANDIDATE: (3, 2) NO_CANDIDATE: (0, 6) revert: (0, 6) [6, 8] = NO_CANDIDATE NO_CANDIDATE: (0, 4) REVERT: (0, 4) [3, 9] = NONE GUESS: (0, 4) [3, 9] = 9 (0, 4) = 9 (5, 4) = 7 (2, 4) = 2 (3, 3) = 9 (3, 5) = 4 (4, 5) = 1 (1, 4) = 3 (3, 2) = 6 (5, 1) = 5 (1, 1) = 6 (1, 2) = 4 (2, 0) = 5 (2, 5) = 6 (4, 1) = 3 (5, 2) = 1 (6, 1) = 2 (0, 5) = 8 (1, 0) = 9 (1, 5) = 5 (2, 3) = 7 (3, 0) = 8 (3, 1) = 7 (3, 8) = 2 (5, 0) = 4 (7, 5) = 3 (1, 8) = 8 (3, 6) = 3 (5, 8) = 6 (6, 5) = 9 (6, 8) = 1 (7, 0) = 1 (0, 6) = 6 (0, 7) = 3 (1, 6) = 2 (2, 8) = 4 (5, 7) = 8 (6, 0) = 3 (7, 6) = 5 (8, 7) = 4 (2, 6) = 1 (4, 6) = 4 (4, 7) = 5 (5, 6) = 9 (6, 6) = 7 (6, 7) = 6 (7, 3) = 6 (7, 7) = 2 (8, 6) = 8 (6, 3) = 5 SOL END

**Correct Solution**

SOL_START (4, 4) = 6 (8, 2) = 5 (4, 3) = 8 (8, 1) = 9 (8, 5) = 7 (0, 1) = 1 (0, 4) = 9 (5, 4) = 7 (2, 4) = 2 (3, 3) = 9 (3, 5) = 4 (4, 5) = 1 (1, 4) = 3 (3, 2) = 6 (5, 1) = 5 (1, 1) = 6 (1, 2) = 4 (2, 0) = 5 (2, 5) = 6 (4, 1) = 3 (5, 2) = 1 (6, 1) = 2 (0, 5) = 8 (1, 0) = 9 (1, 5) = 5 (2, 3) = 7 (3, 0) = 8 (3, 1) = 7 (3, 8) = 2 (5, 0) = 4 (7, 5) = 3 (1, 8) = 8 (3, 6) = 3 (5, 8) = 6 (6, 5) = 9 (6, 8) = 1 (7, 0) = 1 (0, 6) = 6 (0, 7) = 3 (1, 6) = 2 (2, 8) = 4 (5, 7) = 8 (6, 0) = 3 (7, 6) = 5 (8, 7) = 4 (2, 6) = 1 (4, 6) = 4 (4, 7) = 5 (5, 6) = 9 (6, 6) = 7 (6, 7) = 6 (7, 3) = 6 (7, 7) = 2 (8, 6) = 8 (6, 3) = 5 SOL_END

Figure 15: **Example of training data for Sudoku game.** *Top*: initial puzzle setup. *Middle*: full search trace with guesses and backtracking (tabs used). *Bottom*: final correct solution.

**Example of a stacked-CountDown game**

```
Target: 96      Candidates: [22, 77, 24, 48, 31, 12, 36, 35]
```

**DFS Search Trace**

```
Current State: 96:[22, 77, 24, 48, 31, 12, 36, 35], Operations: []
Exploring Operation: 77-48=29, Resulting Numbers: [22, 24, 29, 31, 12, 36, 35]
Generated Node #0,0: 96:[22, 24, 29, 31, 12, 36, 35] Operation: 77-48=29
Moving to Node #0,0
Current State: 96:[22, 24, 29, 31, 12, 36, 35], Operations: ['77-48=29']
Exploring Operation: 22+24=46, Resulting Numbers: [29, 46, 31, 12, 36, 35]
Generated Node #0,0,0: 96:[29, 46, 31, 12, 36, 35] Operation: 22+24=46
Moving to Node #0,0,0
Current State: 96:[29, 46, 31, 12, 36, 35], Operations: ['77-48=29', '22+24=46']
Exploring Operation: 46-29=17, Resulting Numbers: [17, 31, 12, 36, 35]
17,31 unequal
Moving to Node #0,0
...
...
Current State: 96:[29, 2, 31, 12, 36, 35], Operations: ['77-48=29', '24-22=2']
Exploring Operation: 29+2=31, Resulting Numbers: [31, 31, 12, 36, 35]
31,31 equal
Current State: 96:[31, 12, 36, 35], Operations: []
Exploring Operation: 36-35=1, Resulting Numbers: [31, 12, 1]
Generated Node #0,0: 96:[31, 12, 1] Operation: 36-35=1
Moving to Node #0,0
Current State: 96:[31, 12, 1], Operations: ['36-35=1']
Exploring Operation: 31+1=32, Resulting Numbers: [12, 32]
Generated Node #0,0,0: 96:[12, 32] Operation: 31+1=32
Moving to Node #0,0,0
Current State: 96:[12, 32], Operations: ['36-35=1', '31+1=32']
Exploring Operation: 12+32=44, Resulting Numbers: [44]
44,96 unequal: No Solution
...
...
Exploring Operation: 4*24=96, Resulting Numbers: [96]
96,96 equal: Goal Reached
```

**Direct Solution Trace**

```
Current State: 96:[22, 77, 24, 48, 31, 12, 36, 35], Operations: []
Exploring Operation: 77-22=55, Resulting Numbers: [55, 24, 48, 31, 12, 36, 35]
Generated Node #2: [55, 24, 48, 31, 12, 36, 35] from Operation: 77-22=55
Current State: 96:[55, 24, 48, 31, 12, 36, 35], Operations: ['77-22=55']
Exploring Operation: 48-24=24, Resulting Numbers: [55, 24, 31, 12, 36, 35]
Generated Node #3: [55, 24, 31, 12, 36, 35] from Operation: 48-24=24
Current State: 96:[55, 24, 31, 12, 36, 35], Operations: ['77-22=55', '48-24=24']
Exploring Operation: 55-24=31, Resulting Numbers: [31, 31, 12, 36, 35]
31,31 equal
Current State: 96:[31, 12, 36, 35], Operations: []
Exploring Operation: 35-31=4, Resulting Numbers: [4, 12, 36]
Generated Node #2: [4, 12, 36] from Operation: 35-31=4
Current State: 96:[4, 12, 36], Operations: ['35-31=4']
Exploring Operation: 36-12=24, Resulting Numbers: [24, 4]
Generated Node #3: [24, 4] from Operation: 36-12=24
Current State: 96:[24, 4], Operations: ['35-31=4', '36-12=24']
Exploring Operation: 4*24=96, Resulting Numbers: [96]
96,96 equal: Goal Reached
```

Figure 16: **Example of training data for stacked-CoutnDown** (Appendix F). *Top*: game setup. *Middle*: full DFS search trace for training backtracking model. *Bottom*: correct solution path for training direct solution model.