
Supplementary Material for Learning to Better Search with Language Models via Guided Reinforced Self-Training

A Stream of search

Stream of search (SoS) constructs a search tree such that each node at depth d represents a partial solution comprising $d - 1$ reasoning steps and each incoming edge represents a single reasoning step from its parent [2]. A node is considered a leaf when no further reasoning steps can be applied. SoS defines the following primitive tree-search operations expressed in natural language:

- **Generation:** Creates a new child node from the current node.
- **Exploration:** Moves from the current node to one of its child nodes.
- **Backtracking:** Moves to another node when the current node is unpromising.
- **Verification:** Determines whether a leaf node corresponds to a correct solution.

Based on these operations, SoS generates search traces using symbolic search algorithms including depth-first search (DFS) and breadth-first search (BFS). Figure 1 illustrates an example search tree and its corresponding search trace.

A major challenge in training language models on search traces lies in the limitation imposed by their finite context length. Exhaustive search performed by symbolic algorithms often produces traces that exceed the model’s context window, hindering its ability to internalize the complete search process. To address this issue, SoS employs heuristic-guided DFS and BFS to generate shorter traces, albeit at the cost of a lower success rate.

B Countdown benchmark

Each Countdown problem begins with D input numbers and a target number, all of which are integers. The input and target numbers are either one- or two-digit integers. Arithmetic operations are restricted to those that yield non-negative integers. For example, subtraction is only allowed when the larger number is subtracted from the smaller one, and division is permitted only when it results in an integer with no remainder. Since all operations are binary, each operation reduces the number of inputs by one, resulting in a search tree of depth D . Figure 2 shows an example Countdown problem and its search trace generated by a symbolic search algorithm.

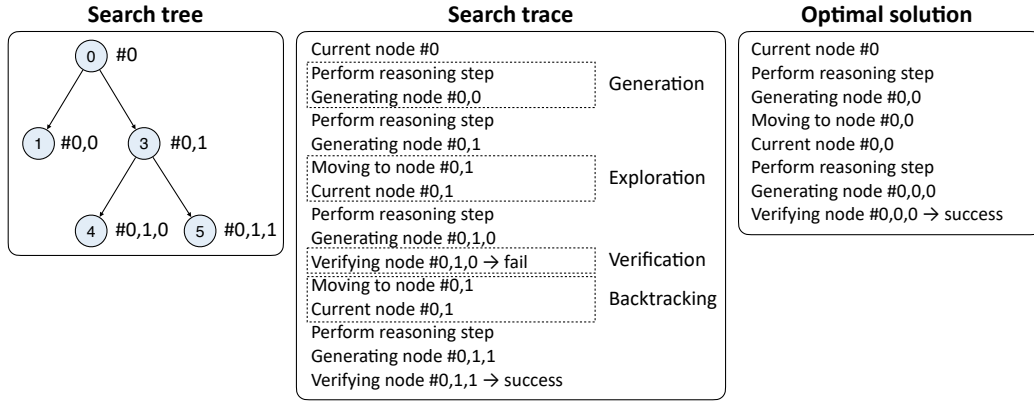


Figure 1: Search tree and its corresponding search trace in SoS. The numbers indicate the order in which nodes are explored.

```

Current State: 25:[56, 58, 15, 8], Operations: []
Exploring Operation: 58-56=2, Resulting Numbers: [15, 8, 2]
Generated Node #0,0: 25:[15, 8, 2] Operation: 58-56=2
Moving to Node #0,0
Current State: 25:[15, 8, 2], Operations: ['58-56=2']
Exploring Operation: 8*2=16, Resulting Numbers: [15, 16]
Generated Node #0,0,0: 25:[15, 16] Operation: 8*2=16
Moving to Node #0,0,0
Current State: 25:[15, 16], Operations: ['58-56=2', '8*2=16']
Exploring Operation: 15+16=31, Resulting Numbers: [31]
31,25 unequal: No Solution
Moving to Node #0,0,0
Current State: 25:[15, 16], Operations: ['58-56=2', '8*2=16']
Exploring Operation: 16-15=1, Resulting Numbers: [1]
1,25 unequal: No Solution
Moving to Node #0,0
Current State: 25:[15, 8, 2], Operations: ['58-56=2']
Exploring Operation: 15*2=30, Resulting Numbers: [8, 30]
Generated Node #0,0,1: 25:[8, 30] Operation: 15*2=30
Moving to Node #0,0,1
Current State: 25:[8, 30], Operations: ['58-56=2', '15*2=30']
Exploring Operation: 30-8=22, Resulting Numbers: [22]
22,25 unequal: No Solution
Moving to Node #0,0,1
Current State: 25:[8, 30], Operations: ['58-56=2', '15*2=30']
Exploring Operation: 8+30=38, Resulting Numbers: [38]
38,25 unequal: No Solution
Moving to Node #0,0
Current State: 25:[15, 8, 2], Operations: ['58-56=2']
Exploring Operation: 15+8=23, Resulting Numbers: [2, 23]
Generated Node #0,0,2: 25:[2, 23] Operation: 15+8=23
Moving to Node #0,0,2
Current State: 25:[2, 23], Operations: ['58-56=2', '15+8=23']
Exploring Operation: 2+23=25, Resulting Numbers: [25]
25,25 equal: Goal Reached

```

Figure 2: Search trace generated by a symbolic algorithm on Countdown.

Search trace	Optimal solution
<p>Current State: 26:[84, 2, 14, 15], Operations: []</p> <p>Exploring Operation: 84/14=6, Resulting Numbers: [2, 15, 6]</p> <p>Generated Node #0,0: 26:[2, 15, 6] Operation: 84/14=6</p> <p>Exploring Operation: 84/2=42, Resulting Numbers: [14, 15, 42]</p> <p>Generated Node #0,1: 26:[14, 15, 42] Operation: 84/2=42</p> <p>Moving to Node #0,0</p> <p>Current State: 26:[2, 15, 6], Operations: ['84/14=6']</p> <p>Exploring Operation: 2*6=12, Resulting Numbers: [15, 12]</p> <p>Generated Node #0,0,0: 26:[15, 12] Operation: 2*6=12</p> <p>Exploring Operation: 15+6=21, Resulting Numbers: [2, 21]</p> <p>Generated Node #0,0,1: 26:[2, 21] Operation: 15+6=21</p> <p>Moving to Node #0,1</p> <p>Current State: 26:[14, 15, 42], Operations: ['84/2=42']</p> <p>Exploring Operation: 42-14=28, Resulting Numbers: [15, 28]</p> <p>Generated Node #0,1,0: 26:[15, 28] Operation: 42-14=28</p> <p>Generated Node #0,1,0: 26:[15, 28] Operation: 42-14=28</p> <p>Exploring Operation: 42-15=27, Resulting Numbers: [14, 27]</p> <p>Generated Node #0,1,1: 26:[14, 27] Operation: 42-15=27</p> <p>Moving to Node #0,0,1</p> <p>Current State: 26:[2, 21], Operations: ['84/14=6', '15+6=21']</p> <p>Exploring Operation: 21-2=19, Resulting Numbers: [19]</p> <p>19,26 unequal: No Solution</p> <p>Exploring Operation: 2+21=23, Resulting Numbers: [23]</p> <p>23,26 unequal: No Solution</p> <p>...</p>	<p>Current State: 26:[84, 2, 14, 15], Operations: []</p> <p>Exploring Operation: 14+15=29, Resulting Numbers: [84, 2, 29]</p> <p>Generated Node #0,0: 26:[84, 2, 29] Operation: 14+15=29</p> <p>Moving to Node #0,0</p> <p>Current State: 26:[84, 2, 29], Operations: ['14+15=29']</p> <p>Exploring Operation: 2*29=58, Resulting Numbers: [84, 58]</p> <p>Generated Node #0,0,0: 26:[84, 58] Operation: 2*29=58</p> <p>Exploring Operation: 84-58=26, Resulting Numbers: [26]</p> <p>26,26 equal: Goal Reached</p>
	<p>Subgoal-augmented trace</p> <p>Current State: 26:[84, 2, 14, 15], Operations: []</p> <p>Exploring Operation: 84/14=6, Resulting Numbers: [2, 15, 6]</p> <p>Generated Node #0,0: 26:[2, 15, 6] Operation: 84/14=6</p> <p>Exploring Operation: 14+15=29, Resulting Numbers: [84, 2, 29]</p> <p>Generated Node #0,1: 26:[84, 2, 29] Operation: 14+15=29</p> <p>Moving to Node #0,0</p> <p>Current State: 26:[2, 15, 6], Operations: ['84/14=6']</p> <p>Exploring Operation: 2*6=12, Resulting Numbers: [15, 12]</p> <p>Generated Node #0,0,0: 26:[15, 12] Operation: 2*6=12</p> <p>Exploring Operation: 15+6=21, Resulting Numbers: [2, 21]</p> <p>Generated Node #0,0,1: 26:[2, 21] Operation: 15+6=21</p> <p>Moving to Node #0,1</p> <p>Current State: 26:[84, 2, 29], Operations: ['14+15=29']</p> <p>...</p>

Figure 3: Search trace generated by the subgoal augmentation algorithm on Countdown. The selected child node (in red) are replace with the corresponding subgoal node (in blue).

C Guided reinforced self-training

Figure 3 illustrates a search trace generated by the subgoal augmentation algorithm on Countdown, where the child node at index (0, 1) is replaced with the corresponding subgoal node at index (0, 0) from the optimal solution. This replacement is performed by modifying the arithmetic operation and resulting numbers in both the generation and exploration steps of the selected child node to match those of the target subgoal node. All operations after the exploration step are discarded, allowing the model to resume the search from the modified node.

D Application to code self-repair

Each search trace in the code self-repair task consists of up to T turns between the model and the user. At the first turn, the model receives a code-generation prompt (Figure 4) that contains the problem description and produces a response with a code solution. The generated code is then executed on the public test set, and the test results are provided as feedback. If all test cases pass, the episode ends. Otherwise, the model receives the self-repair prompt (Figure 5), which includes the test results and a revision instruction, and generates a new response with a revised program.

For Guided-ReST, the model receives a guided prompt (Figure 6) that includes the optimal reference solution at certain turns. However, training directly on this prompt can cause the model to rely on the provided solution rather than internalize it, creating a mismatch between training and inference. To avoid this, we replace the guided prompt with a dummy variant (Figure 7) during training, which preserves the same structure but masks the reference code. This ensures the setup consistent with the inference-time condition and encourages the model to internalize the solution.

```
[user]
Solve the following coding problem using the programming language
python:

{problem}

The input will be given via stdin and the output should be printed to
stdout by your code.

Now solve the problem by providing the code.
```

Figure 4: User prompt for code generation.

```
[user]
Here are the results on the public test cases:

{test_results}

Some test cases are still failing. Please carefully analyze the error
patterns, revise your code to address these issues, and ensure your
solution handles all the test cases correctly. Then, output your final
code.
```

Figure 5: User prompt for code self-repair.

```
[user]
Here are the results on the public test cases:

{test_results}

Some test cases are still failing. Please carefully analyze the error
patterns, revise your code to address these issues, and ensure your
solution handles all the test cases correctly. Then, output your final
code.

Here is the reference code to assist you:

{solution}
```

Figure 6: User prompt for code self-repair with the optimal solution.

```
[user]
Here are the results on the public test cases:

{test_results}

Some test cases are still failing. Please carefully analyze the error
patterns, revise your code to address these issues, and ensure your
solution handles all the test cases correctly. Then, output your final
code.

Here is the reference code to assist you:

```python
REFERENCE CODE HERE
```
```

Figure 7: User prompt for code self-repair with a dummy solution.

Table 1: Training hyperparameters for SoS, ReST, and Guided-ReST.

| Hyperparameter | Value |
|----------------------|-------------|
| The number of epochs | 2 |
| Batch size | 256 |
| Optimizer | AdamW |
| Learning rate | 1e-5 |
| Scheduler | Cosine |
| Adam momentum | [0.9, 0.95] |
| Weight decay | 0.01 |
| Max gradient norm | 1.0 |

Table 2: Training hyperparameters for PPO.

| Hyperparameter | Value |
|--------------------------|-------------|
| The number of epochs | 2 |
| The number of rollouts | 1024 |
| Temperature | 1.0 |
| The number of PPO epochs | 1 |
| PPO clip range | 0.2 |
| Batch size | 256 |
| Discount factor | 1.0 |
| Optimizer | AdamW |
| Actor learning rate | 1e-6 |
| Critic learning rate | 1e-5 |
| Scheduler | Constant |
| Adam momentum | [0.9, 0.95] |
| Weight decay | 0.01 |
| Max gradient norm | 1.0 |

E Implementation details

E.1 Data generation for Countdown

We construct the dataset following the procedure of Gandhi et al. [2]. We use target numbers ranging from 10 to 100 and split them into 90% for training and 10% for testing. We generate 500K examples from the training target number and obtain search trajectories using two heuristic-guided symbolic search algorithms:

- **DFS**: Performs depth-first exploration in increasing order of heuristic values, considering only nodes with heuristic values below the target number.
- **BFS- b** : Performs breadth-first exploration in increasing order of heuristic values, visiting only the b child nodes with the smallest heuristic values for each node. The breadth limit b is set between 1 and 5.

We employ two heuristic functions in conjunction with the search algorithms described above:

- **Sum**: Computes total difference between each input number and the target number.
- **Multiply**: Computes the sum of the minimum differences between each input number and any factor of the target number.

E.2 Hyperparameters

For training, we use a modified implementation of verl [6]. We employ FlashAttention-2 and Liger Kernel to accelerate training [1, 3]. We adopt the default hyperparameter settings for supervised and reinforcement learning provided in the library for both Countdown and code self-repair, with detailed values provided in Tables 1 and 2.

Table 3: Sampling hyperparameters on Countdown.

| Hyperparameter | Value |
|----------------|-------|
| Temperature | 1.0 |
| Top- p | 1.0 |
| Max tokens | 4,096 |

Table 4: Sampling hyperparameters on code self-repair.

| Hyperparameter | Value |
|-----------------|--------|
| Temperature | 1.0 |
| Top- p | 1.0 |
| Max tokens | 16,384 |
| Max turn limits | 4 |

For inference, we use vLLM [4] to achieve high-throughput and memory-efficient response generation. The detailed values of the sampling hyperparameters for Countdown and code self-repair are provided in Tables 3 and 4, respectively.

E.3 Computational resources

We conduct all experiments on an internal HPC cluster, where each node is equipped with two AMD EPYC 7402 CPUs and 750 GB of RAM. We use PyTorch as the base deep learning framework [5]. We use four NVIDIA A100 GPUs for training and a single NVIDIA RTX 3090 GPU for inference.

F Broader impacts

This work aims to enhance the search capabilities of language models, which benefits domains such as code generation and mathematical reasoning by enabling more reliable multi-step decision making. Beyond these domains, improved search capabilities could enhance the reliability and transparency of language models, particularly in applications that require verifiable decision processes such as medicine and scientific discovery. As the method primarily targets arithmetic and symbolic reasoning tasks, it poses minimal risk of misuse or negative societal impact. Future work should explore how improved search capabilities interact with fairness, safety, and interpretability.

References

- [1] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *ICLR*, 2024.
- [2] 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 preprint arXiv:2404.03683*, 2024.
- [3] Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli, Qingquan Song, Shao Tang, Siyu Zhu, Steven Shimizu, Shivam Sahni, Haowen Ning, and Yanning Chen. Liger kernel: Efficient triton kernels for llm training. *arXiv preprint arXiv:2410.10989*, 2024.
- [4] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *SOSP*, 2023.
- [5] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.
- [6] 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.