NOT TOO HARD, NOT TOO EASY: LEARNING FROM INTERMEDIATE STATES FOR LLM STRUCTURED REASONING

Anonymous authors
Paper under double-blind review

ABSTRACT

A common principle of effective learning is to practice material that is neither already mastered nor too difficult to permit progress. We ask how to apply this principle to structured reasoning tasks such as Sudoku and maze solving. In these tasks, a model can repeatedly revise an incomplete or incorrect candidate solution until it satisfies the problem's constraints. The intermediate candidate solutions along this trajectory provide natural training examples: some are already solved, some cannot yet be repaired by the model, and others lie at its current frontier of achievable progress. We therefore investigate whether pretrained language models can learn to revise such states and whether training on states at this frontier improves reasoning more broadly. To achieve this, we couple a pretrained language-model backbone with a recurrent updater that repeatedly revises an explicit solution state, using the same parameters at every update step. We further introduce **F**rontier-**O**riented **C**uration **U**sing **S**elf-trajectories (**FOCUS**), which selects training states from trajectories generated by the current model. FOCUS measures how much the model improves each state within a fixed number of recurrent updates and prioritizes states from which it can make substantial progress. With Qwen3-1.7B, FOCUS achieves 64.4% exact solve accuracy on Sudoku-Extreme and 91.1% on Maze-Hard, with similar gains observed across five Qwen and Llama backbones spanning 1.7B to 8B parameters. We further observe zero-shot transfer in the adapted LLM to mathematical reasoning and code execution, even when the recurrent updater is disabled and no downstream fine-tuning is performed. Code is available here.

1 INTRODUCTION

Learning depends not only on the amount of practice, but also on the difficulty of the material being practiced. Under specific learning assumptions, examples of intermediate difficulty can support faster learning than examples that are too easy or too hard (Wilson et al., 2019). Human learners also use their recent progress to choose what to practice next (Ten et al., 2021). This leads to a basic, yet less investigated, question for reasoning models: *which training examples are most useful at the model's current level of ability?*

We study this question on Sudoku-Extreme and Maze-Hard, two structured reasoning tasks on which prior evaluations report low exact-solve accuracy for language models (Wang et al., 2025; Jolicoeur-Martineau, 2025). Both tasks have explicit solutions that can be checked at every step. A Sudoku grid must satisfy its row, column, and box constraints, while a maze path must connect the start and goal through traversable cells. A model can therefore revise a candidate solution repeatedly, and we can measure whether each revision moves it closer to the correct answer. Because each intermediate solution can be checked, these tasks let us ask which intermediate solutions are most useful for training.

To study this question, we couple a pretrained language model with a recurrent updater that produces a sequence of revisable intermediate states. The language model encodes the problem, and the updater repeatedly revises an explicit answer using the same parameters at every step. Each intermediate state contains the current answer and the updater's memory. Standard training begins from the initial state.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

We can also save an intermediate state and resume training from it. The question then becomes not only which problems to use, but also *where in each solution sequence to resume training*.

Prior approaches use several sources of training states. Training can begin from the initial state used at inference, an artificially corrupted target, or a state visited by the current model. However, these options do not explain which state within a model-generated sequence is most useful for additional training.

We address this problem with Frontier-Oriented Curation Using Self-trajectories (FOCUS). During training, FOCUS records the states visited by the current solver. From each candidate state, it measures the proportional reduction in task error over a fixed number of recurrent steps. It then selects the state whose reduction is closest to a chosen target. Here, difficulty is defined by short-term repair progress: states that improve much more than the target are relatively easy, states that improve much less are relatively hard, and states near the target form the *repair frontier*. Because FOCUS measures progress using the current solver, the frontier changes as the model improves. Figure 1 summarizes the procedure.

After selecting a state, FOCUS resumes training from both its saved answer and its saved memory. The training loss penalizes the model when its error reduction falls short of the target. For compatible updates that start from the original state, a frozen TRM teacher also provides soft prediction targets at selected steps (Hinton et al., 2015). At inference, the language model encodes the problem once, the updater runs for a fixed number of steps, and the model decodes only the final state. State selection, teacher predictions, and training-time error scores are not used at inference.

With Qwen3-1.7B, FOCUS solves 64.4% of Sudoku-Extreme puzzles and 91.1% of Maze-Hard instances exactly. Compared with Recurrent final-only, which trains the recurrent model using only final-state supervision, FOCUS improves accuracy by 57.2 and 5.1 percentage points, respectively. FOCUS also improves over this baseline across five Qwen and Llama backbones ranging from 1.7B to 8B parameters. Table 1 summarizes these results, and Appendix D.1 provides further evaluation details.

We also test whether language-model parameters learned during Sudoku training also improve reasoning performance on mathematical and code-reasoning tasks. To do so, we disable the recurrent updater and evaluate the adapted language model on these tasks *without additional training*. Compared with the original Qwen3-1.7B model, the Sudoku-trained adapter improves zero-shot accuracy from 72.36% to 72.96% on MATH-Hard and from 50.75% to 55.38% on CruxEval-O.

Our contributions are threefold:

- We show how a pretrained language model can guide a recurrent updater that repeatedly revises an explicit answer. We compare direct prediction, one update, and repeated updates across two structured tasks and five language-model backbones.
- We introduce FOCUS, which selects intermediate states by measuring how much the current model improves them over a fixed number of steps. Training resumes from the selected answer and memory and penalizes progress below a chosen target.
- We provide a controlled evaluation of whether FOCUS changes the language model beyond its training tasks. We show that language-model adapters trained with FOCUS can improve zero-shot performance on different mathematical and code-reasoning benchmarks.

2 Related Work

Iterative reasoning. Prior work develops specialized recurrent architectures for structured reasoning. The Hierarchical Reasoning Model (HRM) uses recurrent modules operating at different timescales, while the Tiny Recursive Model (TRM) uses a smaller recursive network to revise solutions repeatedly (Wang et al., 2025; Jolicoeur-Martineau, 2025). The Denoising Recursion Model (DRM) trains a recursive network to recover correct solutions from deliberately corrupted ones (Cameron et al., 2026). These models demonstrate the value of repeated solution updates, but they focus on model architecture and repair training rather than selecting states from the current updater's trajectory based on measured progress. Unlike these specialized recurrent architectures, FOCUS couples a task-specific recurrent updater with a pretrained language model and is compatible with multiple language-model backbones.

Training-state selection. Prior methods use loss (Shrivastava et al., 2016), predefined difficulty (Parashar et al., 2026), success rates (Florensa et al., 2017; 2018), or value-estimation errors (Jiang et al., 2021) to decide what the model should practice. These signals select examples, starting states, goals, or environments, but they do not directly measure how repeated updates change a particular intermediate solution. As such, initial error is insufficient for this purpose because two states with similar errors may respond differently to the same updates. FOCUS instead runs the current updater for a fixed number of steps from each candidate state and measures the proportional reduction in task error. An observed reduction larger than the target exceeds the desired progress, whereas a smaller reduction falls short. FOCUS selects the state whose reduction is closest to this target, called the repair frontier, rather than selecting by initial error or by the largest observed improvement.

3 METHOD

Overview. Figure 1 summarizes FOCUS. A pretrained language model encodes each problem, and a recurrent updater uses this representation to revise an explicit answer over successive steps. Training from the task-defined initial state produces a trajectory of intermediate states. FOCUS selects one state on this trajectory, restores both its answer and recurrent memory, and trains the updater to continue from it. The selection criterion measures how much the current updater reduces task error over a fixed number of steps and chooses the state whose reduction is closest to a prescribed target. Training from the original initialization remains part of the procedure. At inference, the updater starts from this initial state and applies a fixed number of updates.

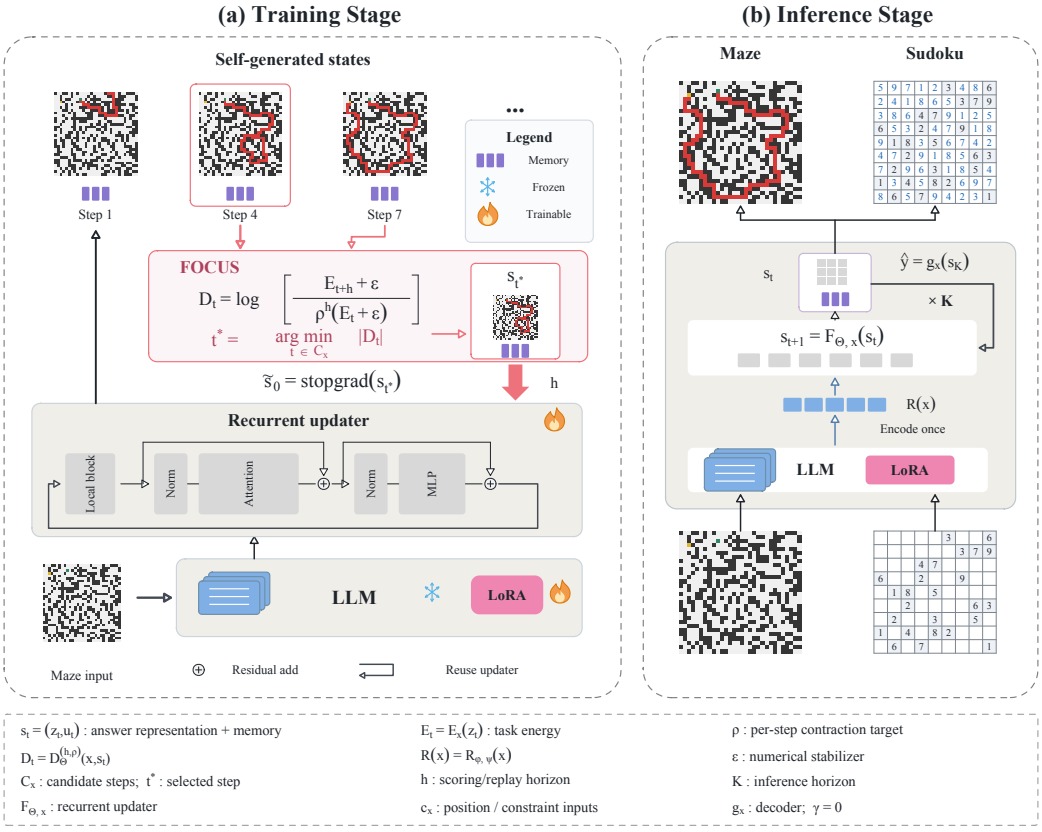

Figure 1: Overview of FOCUS. (a) Training selects intermediate states near the repair frontier and resumes recurrent updates from the selected answer and memory. (b) At inference, the adapted LLM encodes the input once to condition K recurrent updates, followed by a single final decode.

3.1 LANGUAGE-MODEL-CONDITIONED RECURRENT UPDATES

For an input x, the model maintains a state $s_t = (z_t, u_t)$. The component z_t is the current answer, and u_t is the memory carried between updates. Training uses the correct answer $y^\star$. A pretrained language model encodes x, and a task-specific projection maps the encoding to

$$R_{\phi,\psi}(x) = G_\psi\big(\mathrm{LM}_{W_0+\Delta W_\phi}(x)\big)\,. \tag{1}$$

The pretrained weights W_0 remain fixed. The low-rank adaptation parameters ϕ and the projection parameters ψ are trained (Hu et al., 2022). The representation $R_{\phi,\psi}(x)$ supplies the problem context, while s_t records the answer being revised. It is computed once and reused at every update.

The trainable parameters are $\Theta = (\theta, \phi, \psi)$. A recurrent updater, separate from the language model, computes

$$s_{t+1} = F_{\Theta,x}(s_t) := F_\theta(s_t; R_{\phi,\psi}(x), c_x)\,, \tag{2}$$

where c_x contains deterministic positional and task-constraint inputs. The updater parameters θ are shared across steps. Gradients from these updates reach θ, ϕ, and ψ.

At inference, the language model encodes x once. The updater starts from $s_0^{\mathrm{clean}}(x)$, which is independent of $y^\star$, and applies K updates. The prediction is $\hat{y} = g_x(s_K)$. Appendix C specifies the task-specific states and initialization, and Appendix D specifies decoding.

3.2 FRONTIER-BASED STATE SELECTION

Measuring repair behavior. A state's current error indicates how much work remains. Its continuation shows how the updater handles that work. States with similar error can improve at different rates under the same number of updates. FOCUS therefore measures energy reduction relative to starting energy over a fixed horizon.

Candidate states. With gradient tracking disabled, the current updater generates a trajectory $\tau(x) = (s_0^{\mathrm{clean}}, s_1, \ldots, s_{K_p})$. Here K_p is the collection horizon. Scoring a state s_t requires its state after h updates. Both states appear in the collected trajectory when $0 \le t \le K_p - h$. The candidate set $\mathcal{C}_x$ contains the indices in this range that also satisfy the task-specific eligibility rules in Appendix C.1. That appendix also specifies tie-breaking. The horizon h is used for both scoring and replay, while K is the inference horizon. Parameters remain fixed during collection and scoring.

Contraction defect. Let $E_x(z) \ge 0$ be a task energy combining supervised error with task-specific penalties, as defined in Appendix B. Lower energy indicates improvement under this surrogate. It may use $y^\star$, so self-generated states do not imply self-generated supervision. For $s = (z, u)$, write $z^{(h)} = \pi_z F^h_{\Theta,x}(s)$, where π_z extracts the answer component. We define

$$D^{(h,\rho)}_\Theta(x, s) = \log\left(\frac{E_x(z^{(h)}) + \epsilon}{\rho^h\big(E_x(z) + \epsilon\big)}\right), \qquad 0 < \rho < 1, \quad \epsilon > 0. \tag{3}$$

The reference factor ρ^h specifies a target reduction over h steps, not a rate assumed to hold for the learned updater. A nonpositive defect meets this target for the smoothed energy $E_x + \epsilon$; a positive defect falls short, even when energy decreases. The logarithm expresses multiplicative deviations as signed additive quantities, which accumulate across consecutive windows (*cf.* Appendix A).

Selecting the repair frontier. Defects near zero identify continuations near the target reduction. Small positive values indicate a small shortfall, while small negative values indicate that the target has just been met. FOCUS selects from either side of this boundary. For a nonempty candidate set, it chooses

$$t^\star = \operatorname*{arg\,min}_{t \in \mathcal{C}_x} \Big| D^{(h,\rho)}_\Theta(x, s_t) - \gamma \Big|\,. \tag{4}$$

We use $\gamma = 0$ and call states near this level the *repair frontier*. The closest eligible state is selected even if its defect is far from zero. As the updater changes, different states can become closest to the target.

3.3 Replaying Complete Recurrent States

Restoring the selected state. A continuation depends on both the answer z_t and memory u_t. FOCUS restores the complete selected state and trains an h-step continuation

$$\widetilde{s}_0 = \mathrm{stopgrad}(s_{t^\star}), \qquad \widetilde{s}_{j+1} = F_{\Theta,x}(\widetilde{s}_j), \quad j = 0, \ldots, h-1. \tag{5}$$

The saved state is detached, so backpropagation covers the replay continuation. The conditioning representation is recomputed with gradient tracking. Replay can therefore update θ, ϕ, and ψ.

Task supervision and local reduction. Task supervision trains the replayed continuation toward the correct answer. An additional term penalizes shortfalls relative to the selected state's contraction target. Let D^{rep} denote (3) recomputed on the gradient-tracked replay. The replay objective is

$$\mathcal{L}_{\mathrm{aux}}^{\mathrm{FOCUS}} = \mathcal{L}_{\mathrm{task}}(\widetilde{s}_{1:h}; x, y^\star) + \beta_{\mathrm{def}}[D^{\mathrm{rep}} - \gamma]_+^2, \tag{6}$$

where $[a]_+ = \max(a, 0)$ and $\beta_{\mathrm{def}} \geq 0$. The one-sided penalty applies when replay falls short of the target. Task supervision applies throughout the replay continuation. Gradients through the defect term follow the replayed updates. The task loss and energy may have different components and temporal reductions, as specified in Appendix C.

Training from the original initialization. Replay trains continuations from visited states. We also train rollouts from the task-defined initial state used at evaluation. On compatible teacher-supervised initial-state updates, temperature-scaled KL supervision matches the student's soft predictions to those of a frozen TRM teacher at aligned recurrent steps. Teacher supervision applies to initial-state updates.

Sudoku includes a base loss in every mini-batch and adds replay loss when scheduled. Maze-Hard alternates between initial-state and replay updates. When $\mathcal{C}_x$ is empty, Sudoku omits the replay term and Maze-Hard uses an initial-state update. Branch weights, teacher alignment, the Fixed Mix baseline, and the complete algorithm are specified in Appendix C. At inference, the model follows the fixed rollout defined in Section 3.1.

4 Experiments

We first evaluate how FOCUS can solve Sudoku-Extreme and Maze-Hard problems. Then we examine zero-shot transfer performance on code and math tasks, followed by a discussion of training-state curation, ablations, and qualitative analysis of recurrent trajectories.

4.1 Experimental Setup

Tasks and data. Sudoku-Extreme uses an 81×9 answer-logit grid with given digits clamped after each update (Jolicoeur-Martineau, 2025). Training uses symmetry augmentations of 707 source puzzles, supplemented with synthetic puzzles. Both training corpora are puzzle-disjoint from the 1,000 development and 1,000 held-out test problems. Maze-Hard uses $900/100/1{,}000$ train/development/test mazes. Appendix D provides the data construction details and output formats.

Models and training configurations. We evaluate Qwen3-1.7B, Qwen3-4B, Qwen3-8B, Llama-3.2-3B-Instruct, and Llama-3.1-8B-Instruct (Yang et al., 2025; Meta, 2024; Grattafiori et al., 2024). Base model uses the pretrained language model without task-specific training. Direct LoRA SFT trains the model to generate answers directly. One-step State-SFT supervises a single explicit-state update. Recurrent final-only supervises the final answer after repeated shared updates. FOCUS follows the training procedure in Section 3. All experiments used a fixed seed of 42. The curation study uses Qwen3-1.7B. Teacher-supported configurations apply TRM distillation to compatible initial-state updates. Appendix D.2 gives more detailed training configurations.

Metrics. In Sudoku and Maze, a problem counts as solved only when the entire predicted answer matches the correct answer. We use the validation dataset to select checkpoints and, for the depth analysis, the number of updates. We report final accuracy on a separate, held-out test set. At test time, the model starts from the task-defined initial state, makes a fixed number of updates, and produces

Table 1: **Source-task accuracy across language-model backbones.** Exact solve accuracy (%) on Sudoku-Extreme and Maze-Hard.

Backbone	Sudoku-Extreme				Maze-Hard			
	Direct	One-step	Recurrent	**FOCUS**	Direct	One-step	Recurrent	**FOCUS**
Qwen3-1.7B	0.5	0.5	7.2	**64.4**	0.0	0.0	86.0	**91.1**
Qwen3-4B	0.5	0.3	16.2	**66.0**	0.1	0.0	86.9	**91.5**
Qwen3-8B	0.6	0.7	14.8	**64.7**	0.0	0.0	87.2	**88.3**
Llama-3.2-3B-Instruct	0.0	0.4	12.3	**61.9**	0.0	0.0	86.8	**90.1**
Llama-3.1-8B-Instruct	0.3	0.8	8.0	**65.8**	0.0	0.0	79.8	**87.7**

Direct: Direct LoRA SFT; One-step: One-step State-SFT; Recurrent: Recurrent final-only. All unadapted base models score 0.0% on both tasks.

one answer. Each problem receives one attempt without search, teacher assistance, or feedback from a checker. Scoring and selecting intermediate states, then training from them, take place only during training.

4.2 Results

Table 1 reports results on Sudoku-Extreme and Maze-Hard. With Qwen3-1.7B, FOCUS achieves 64.4% on Sudoku-Extreme and 91.1% on Maze-Hard, compared with 7.2% and 86% for Recurrent final-only. Direct LoRA SFT and One-step State-SFT each achieve 0.5% on Sudoku-Extreme and 0 accuracy on Maze-Hard. Repeated updates also improve solving performance over direct and one-step prediction on the other backbones. Table 2 shows that all three prompt-only reasoning models achieve 0 accuracy on both tasks. Appendix D.1 provides evaluation details.

Table 2: **Prompt-only reasoning models on source tasks.** Exact solve accuracy (%).

Model	Sudoku-Extreme	Maze-Hard
DeepSeek-R1	0.0	0.0
Claude 3.7 8K	0.0	0.0
o3-mini (high)	0.0	0.0

In sum, training recurrence using only final-answer supervision solves most Maze instances but performs substantially worse on Sudoku. In comparison, FOCUS achieves higher accuracy on both tasks. Section 4.3 further discusses curation details to explain this difference.

4.3 Zero-Shot Transfer

We disable the recurrent module and evaluate the language-model adapters trained on the source tasks using zero-shot CoT prompts and one greedy completion per item. Evaluation uses neither worked demonstrations nor downstream parameter updates (Kojima et al., 2022). Table 4 reports results across all backbones. Appendix D.4 describes checkpoint selection and downstream evaluation settings.

For Qwen3-1.7B, FOCUS (Sudoku) raises MATH-Hard accuracy from 72.36% to 72.96%, CruxEval-O from 50.75% to 55.38%, and CruxEval-I from 24% to 26.75% (Hendrycks et al., 2021; Gu et al., 2024).

Table 3: Zero-shot transfer to AIME25 and AIME26 after structured-repair training.

Tasks	AIME25	AIME26
Qwen3-1.7B		
Baseline	23.33	16.67
FOCUS (Sudoku)	16.67	**23.33**
FOCUS (Maze)	**26.67**	13.33
Qwen3-4B		
Baseline	20.00	30.00
FOCUS (Sudoku)	26.67	**36.67**
FOCUS (Maze)	**30.00**	33.33
Qwen3-8B		
Baseline	26.67	30.00
FOCUS (Sudoku)	20.00	30.00
FOCUS (Maze)	**33.33**	**33.33**

In settings-matched comparisons, these changes occur without task-specific recurrent computation at evaluation and therefore measure transfer through the adapted language model. MATH-Hard is the 1,324 problem Level-5 subset of MATH (Hendrycks et al., 2021; LightEval, 2024), and AIME25

(Test-Time Compute Organization, 2025) and AIME26 (Dekoninck et al., 2026) each contain 30 problems, so small percentage differences on those competition sets should be read in light of their sample sizes. Table 3 reports the AIME results, which vary across backbones and source tasks.

Table 4: Zero-shot transfer after structured-repair training. Accuracy (%) with one greedy completion per item. The recurrent module is disabled and no downstream parameters are updated.

	Mathematics		Code execution		
Source-task adaptation	**MATH**	**MATH-Hard**	**CruxEval-O**	**CruxEval-I**	**Avg.**
Qwen3-1.7B					
Baseline	**86.54**	72.36	50.75	24.00	58.41
FOCUS (Sudoku)	86.38	**72.96**	**55.38**	**26.75**	**60.37**
FOCUS (Maze)	86.36	71.68	50.63	24.12	58.20
Qwen3-4B					
Baseline	91.58	83.46	79.63	45.25	74.98
FOCUS (Sudoku)	90.92	82.40	79.50	44.25	74.27
FOCUS (Maze)	**92.12**	**83.91**	**79.75**	**45.50**	**75.32**
Qwen3-8B					
Baseline	91.14	82.48	81.63	**57.75**	78.25
FOCUS (Sudoku)	91.48	82.40	**82.38**	55.38	77.91
FOCUS (Maze)	**91.52**	**85.20**	82.13	54.88	**78.43**
Llama-3.2-3B-Instruct					
Baseline	**47.40**	21.90	15.25	0.00	21.14
FOCUS (Sudoku)	45.28	21.15	15.88	**0.38**	20.67
FOCUS (Maze)	**47.40**	**22.43**	**16.88**	0.25	**21.74**
Llama-3.1-8B-Instruct					
Baseline	49.16	22.66	**27.00**	21.75	30.14
FOCUS (Sudoku)	**49.96**	22.81	26.75	21.50	30.26
FOCUS (Maze)	49.86	**23.49**	26.75	**22.00**	**30.53**

Zero-shot CoT prompting uses no worked examples. Baseline is the unadapted backbone; FOCUS (Sudoku) and FOCUS (Maze) use its language-model LoRA adapter after training on the named source task. Avg. is the unweighted mean over the four benchmarks. Bold marks the nonzero maximum within each backbone and column.

We study three design choices in FOCUS: how replay states are selected, whether local repair progress is regularized, and whether replay restores the recurrent memory associated with a selected state. Table 5 compares complete curation schemes. The remaining ablations change one component of FOCUS while holding the rest of the training recipe fixed.

Complete curation schemes. Table 5 compares Random, Energy-Hard, Fixed Mix, and FOCUS with Qwen3-1.7B at $K = 128$ on Sudoku-Extreme and $K = 16$ on Maze-Hard. The schemes differ in the additional states used for training. FOCUS selects states visited by the model's own rollout near the repair frontier and uses the defect penalty; the other rows use their respective curation rules.

Table 5: State-curation comparison with Qwen3-1.7B. Held-out exact solve accuracy (%) on Sudoku-Extreme ($K = 128$) and Maze-Hard ($K = 16$).

Method	**Replay-state selection**	**Sudoku-Extreme**	**Maze-Hard**
Random	Uniformly sampled rollout state	57.8	89.7
Energy-Hard	Rollout state with the highest task error	61.3	89.9
Fixed Mix	Fixed mixture of training-state types	60.8	90.5
FOCUS	Rollout state nearest the repair frontier	**64.4**	**91.1**

Sudoku-Extreme uses $K = 128$ and Maze-Hard uses $K = 16$. Bold denotes the numerical maximum in each task.

FOCUS attains the highest accuracy in both columns: 64.4% on Sudoku-Extreme and 91.1% on Maze-Hard. Fixed Mix is the strongest alternative on Maze-Hard at 90.5%, while Energy-Hard is the strongest alternative on Sudoku-Extreme at 61.3%. Thus, the complete

FOCUS training scheme outperforms the other curation schemes under the reported seed-42 settings. The next two ablations identify the contributions of its objective and state-selection rule.

Table 6: **Defect-penalty ablation.** Qwen3-1.7B exact solve accuracy (%) with and without the penalty.

Method	Sudoku-Extreme	Maze-Hard
FOCUS w/o defect penalty	61.3	85.8
FOCUS	**64.4**	**91.1**

Both variants retain frontier selection and full-state replay. Bold marks the higher accuracy in each task.

Defect penalty. We retain frontier selection and replay of the complete recurrent state while setting $\beta_{\text{def}} = 0$. On Maze-Hard, removing the penalty lowers exact accuracy from 91.1% to 85.8%. The 5.3-percentage-point decrease shows that the penalty makes a substantial contribution to the complete FOCUS objective in this configuration.

Frontier versus random selection. We isolate FOCUS's selection rule by replacing frontier selection with uniform sampling from the same eligible rollout states. The control retains the defect penalty, replay of the complete recurrent state, training budget, and evaluation settings; it differs only in how the replay state is chosen. On the held-out Maze-Hard test set at $K = 16$, full FOCUS achieves 91.1% exact accuracy, whereas uniform selection achieves 87.1%. Thus, selecting states at the repair frontier improves exact solve accuracy by 4.0 percentage points in this seed-42 experiment. This comparison identifies the frontier rule, rather than the defect penalty or memory-preserving replay, as the source of this improvement.

4.4 Recurrent Repair Trajectories

Figure 2 follows a held-out example from each task. In Maze-Hard, the predicted path first matches the reference at $t = 16$. In Sudoku-Extreme, the displayed predictions contain 29, 23, 16, 11, and then zero incorrect cells, with the first exact grid at $t = 34$. The rightmost panel in each row is the target. Longer repair trajectories are shown in Appendices E and F.

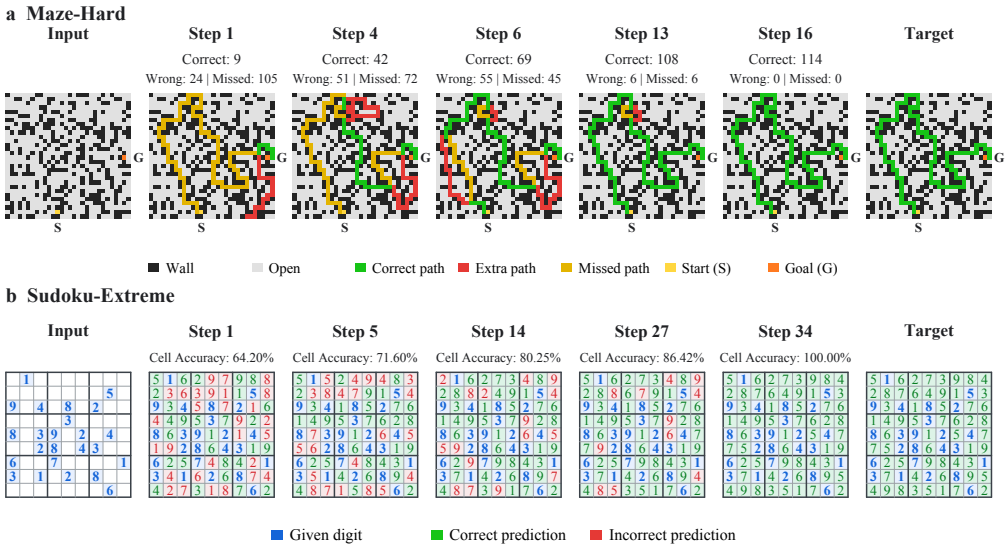

Figure 2: Recurrent repair with Qwen3-1.7B FOCUS. (a) The Maze-Hard example first matches the target at step 16. (b) The Sudoku-Extreme example first matches the target at step 34. Maze path counts exclude the start and goal; Sudoku cell accuracy includes all 81 cells.

Comparison with baselines. Figure 3 compares predictions on the same Maze-Hard test instance. Direct LoRA SFT and One-step State-SFT predict extra path cells and omit parts of the reference path. Recurrent final-only progressively forms a valid route but retains a detour at step 16. FOCUS removes the extra path cells and recovers the missing segments, matching the shortest reference path at step 7 and preserving it through step 16. The comparison shows how FOCUS refines an incomplete path into an exact solution, while the baselines leave incorrect cells or an unnecessary detour.

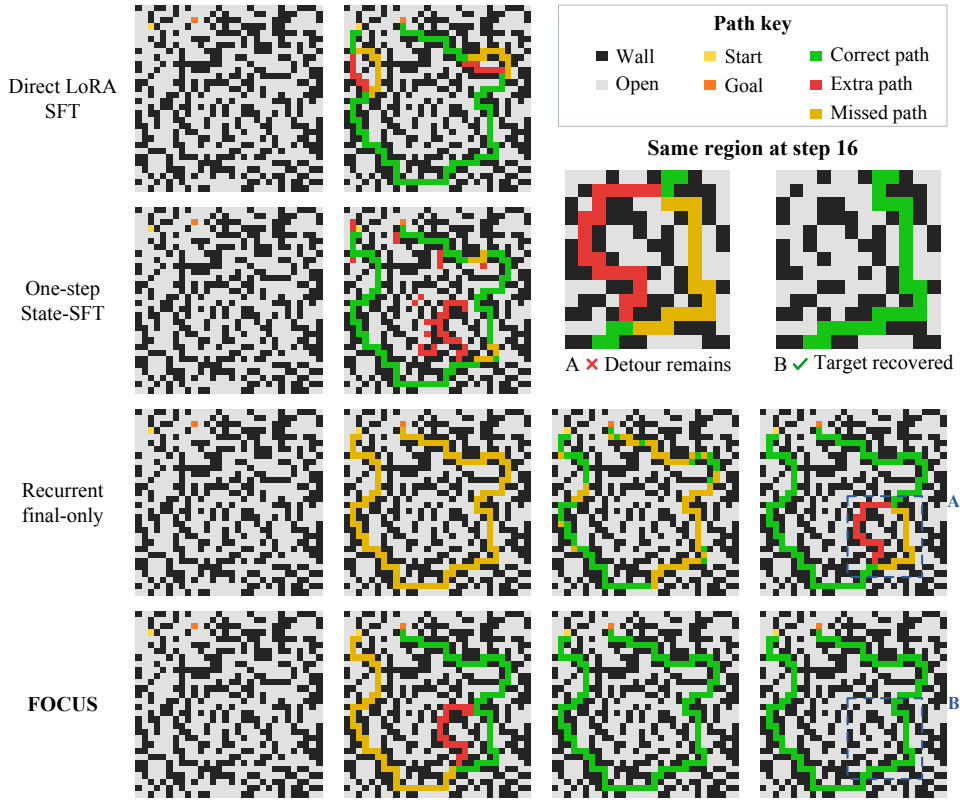

Figure 3: Maze-Hard predictions with Qwen3-8B. Each row starts with the input. Direct and One-step show final predictions; Recurrent final-only and FOCUS show steps 4, 7, and 16. The recurrent baseline ends with a valid but non-shortest path; FOCUS matches the reference at step 7 and retains it at step 16. Insets A and B magnify the boxed regions at step 16. Colors indicate agreement with the reference after inference.

5 Conclusion

We studied whether pretrained language models can support iterative explicit-state repair and how the states used to train such a solver should be chosen. Coupling an adapted language model with a shared recurrent updater improves structured solving over direct prediction and a single update in the evaluated configurations. FOCUS selects states from the model's own rollouts according to short-horizon repair progress, then replays the complete answer–memory state; the curation and controlled ablation results support this training design on Maze-Hard. When the recurrent updater is removed, the adapted language-model parameters retain zero-shot gains on mathematical and code-reasoning benchmarks. These findings position structured repair tasks as a useful setting for studying both iterative solving and how task-specific training can shape reusable language-model representations.

Reproducibility Statement

The supplementary material provides the implementation, configuration files, and scripts used to train and evaluate FOCUS and the reported baselines. Section 3, Appendix C, and Table 7 specify the architecture, objective, state-selection procedure, optimization settings, and task-specific schedules. Section 4 and Appendix D define the data splits, development selection rules, deterministic decoding procedures, and exact-solve metrics; Appendix D.4 documents the zero-shot prompts and downstream evaluation protocol. The experiments use publicly available pretrained Qwen and Llama backbones and public downstream benchmarks. We provide the fixed held-out evaluation splits, random seeds, checkpoints, and per-instance predictions needed to reproduce the reported source-task and transfer results.

AI Use Statement

In preparing this paper, we used Large Language Models only to aid with language polishing, grammar refinement, and improving readability. All research ideas, methodological contributions, experimental design, and analysis were conceived and carried out entirely by the authors. The LLM was not involved in ideation, technical writing of results, or scientific claims. The authors take full responsibility for all content of this paper.

References

Chris Cameron, Wangzheng Wang, Nikita Ivanov, Ashmita Bhattacharyya, Didier Chételat, and Yingxue Zhang. One step forward and k steps back: Better reasoning with denoising recursion models. In *Workshop on Latent & Implicit Thinking – Going Beyond CoT Reasoning*, 2026. URL https://openreview.net/forum?id=o1KnUEh8Nj.

Jasper Dekoninck, Nikola Jovanović, Tim Gehrunger, Kári Rögnvaldsson, Ivo Petrov, Chenhao Sun, and Martin Vechev. Beyond benchmarks: Matharena as an evaluation platform for mathematics with LLMs. In *3rd AI for Math Workshop: Toward Self-Evolving Scientific Agents*, 2026. URL https://openreview.net/forum?id=DmPE4byHuN.

Carlos Florensa, David Held, Markus Wulfmeier, Michael Zhang, and Pieter Abbeel. Reverse curriculum generation for reinforcement learning. In Sergey Levine, Vincent Vanhoucke, and Ken Goldberg (eds.), *Proceedings of the 1st Annual Conference on Robot Learning*, volume 78 of *Proceedings of Machine Learning Research*, pp. 482–495. PMLR, 13–15 Nov 2017. URL https://proceedings.mlr.press/v78/florensa17a.html.

Carlos Florensa, David Held, Xinyang Geng, and Pieter Abbeel. Automatic goal generation for reinforcement learning agents. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1515–1528. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/florensa18a.html.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie

Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani,

Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 16568–16621. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/gu24c.html.

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. In J. Vanschoren and S. Yeung (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/be83ab3ecd0db773eb2dc1b0a17836a1-Paper-round2.pdf.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015. URL https://arxiv.org/abs/1503.02531.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Minqi Jiang, Edward Grefenstette, and Tim Rocktäschel. Prioritized level replay. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 4940–4950. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/jiang21b.html.

Alexia Jolicoeur-Martineau. Less is more: Recursive reasoning with tiny networks, 2025. URL https://arxiv.org/abs/2510.04871.

Takeshi Kojima, Shixiang (Shane) Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 22199–22213. Curran Associates, Inc., 2022. doi: 10.52202/068431-1613. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/8bb0d291acd4acf06ef112099c16f326-Paper-Conference.pdf.

LightEval. MATH-Hard dataset. https://huggingface.co/datasets/lighteval/MATH-Hard, 2024. Accessed 2026-09-03.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

Meta. Llama-3.2-3B-Instruct model card, 2024. URL https://huggingface.co/meta-llama/Llama-3.2-3B-Instruct. Accessed 2026-09-03.

Fausto Milletari, Nassir Navab, and Seyed-Ahmad Ahmadi. V-Net: Fully convolutional neural networks for volumetric medical image segmentation. In *2016 Fourth International Conference on 3D Vision (3DV)*, pp. 565–571, 2016. doi: 10.1109/3DV.2016.79.

Shubham Parashar, Shurui Gui, Xiner Li, Hongyi Ling, Sushil Vemuri, Blake Olson, Eric Li, Yu Zhang, James Caverlee, Dileep Kalathil, and Shuiwang Ji. Curriculum reinforcement learning from easy to hard tasks improves LLM reasoning. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=KJvHnl3kUv.

Abhinav Shrivastava, Abhinav Gupta, and Ross Girshick. Training region-based object detectors with online hard example mining. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 761–769, June 2016. doi: 10.1109/CVPR.2016.89.

Alexandr Ten, Pramod Kaushik, Pierre-Yves Oudeyer, and Jacqueline Gottlieb. Humans monitor learning progress in curiosity-driven exploration. *Nat. Commun.*, 12(1):5972, October 2021. doi: 10.1038/s41467-021-26196-w.

Test-Time Compute Organization. AIME 2025: Unified test-time scaling format. https://huggingface.co/datasets/test-time-compute/aime_2025, 2025. Accessed 2026-09-03.

Guan Wang, Jin Li, Yuhao Sun, Xing Chen, Changling Liu, Yue Wu, Meng Lu, Sen Song, and Yasin Abbasi Yadkori. Hierarchical reasoning model, 2025. URL https://arxiv.org/abs/2506.21734.

Robert C Wilson, Amitai Shenhav, Mark Straccia, and Jonathan D Cohen. The eighty five percent rule for optimal learning. *Nat. Commun.*, 10(1):4646, November 2019. doi: 10.1038/s41467-019-12552-4.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

A Proof of the Finite-Horizon Energy Bound

We use the input-conditioned transition $F_{\Theta,x}$ defined in Section 3.1.

Let $s_{b+1} = F^h_{\Theta,x}(s_b)$ for $b = 0, \dots, B-1$, and write $E_b = E_x(\pi_z s_b)$. From (3),

$$D_b = \log(E_{b+1} + \epsilon) - \log(E_b + \epsilon) - h \log \rho. \tag{7}$$

Summing over blocks telescopes:

$$\log(E_B + \epsilon) = \log(E_0 + \epsilon) + Bh \log \rho + \sum_{b=0}^{B-1} D_b. \tag{8}$$

Since $D_b \le [D_b]_+$, exponentiation yields the finite-horizon bound, with $K = Bh$:

$$E_x(z_K) + \epsilon \le (E_x(z_0) + \epsilon)\, \rho^K \exp\left(\sum_{b=0}^{B-1} [D_b]_+ \right). \tag{9}$$

The bound follows from an exact accounting identity for task-energy change; it is not a global convergence theorem for the neural transition.

For a tolerance $\delta > 0$, a conceptual frontier band can be written as

$$\mathcal{F}_{\theta,\gamma,\delta}(x) = \{s : |D^{(h,\rho)}_\theta(x, s) - \gamma| \le \delta\}.$$

The implemented selector minimizes distance to γ over eligible candidates. The tolerance describes the band; it does not introduce an additional acceptance gate or alter the selected candidate.

B Task-Energy Definitions

B.1 Sudoku

Let $p_{i,d} = \mathrm{softmax}(z_i)_d$. A practical energy is

$$E^{\mathrm{Sudoku}}_x(z) = \lambda_{\mathrm{CE}} \ell_{\mathrm{CE}}(p, y^\star) + \lambda_{\mathrm{hard}} \ell_{\mathrm{hard}}(p, y^\star) + \lambda_c V_{\mathrm{allDiff}}(p), \tag{10}$$

where

$$V_{\mathrm{allDiff}}(p) = \frac{1}{27 \cdot 9} \sum_{g \in \mathcal{G}} \sum_{d=1}^{9} \left(\sum_{i \in g} p_{i,d} - 1 \right)^2, \tag{11}$$

with $\mathcal{G}$ containing nine rows, nine columns, and nine boxes. The hard-cell term is the mean cross-entropy over the highest-loss fraction of non-given cells. Both cross-entropy terms are restricted to non-given cells, and givens are clamped after every update.

B.2 Maze-Hard

Let p_i be the probability that open cell i belongs to the path. A practical energy is

$$E^{\mathrm{Maze}}_x(z) = \ell^{\mathrm{open}}_{\mathrm{CE}}(p, y^\star) + \lambda_{\mathrm{dice}} \ell_{\mathrm{Dice}}(p, y^\star) + \lambda_{\mathrm{rank}} V_{\mathrm{rank}}(p) + \lambda_{\mathrm{dec}} V_{\mathrm{dec}}(p). \tag{12}$$

Here ℓ_{Dice} is a soft Dice loss (Milletari et al., 2016). Here V_{rank} is a top-k path/non-path boundary-margin penalty, and V_{dec} is a differentiable surrogate for the fixed decoder. Its task-specific components and weights are specified in the released configuration files. Exact path validity is computed only after decoding and is not fed back during evaluation.

C Implementation Details

This appendix specifies the training protocol used for the recurrent explicit-state solver and the two complete state-curation schemes evaluated in the main text. The recurrent architecture is shared across the curation methods. Fixed Mix and FOCUS differ in how intermediate training states are constructed and selected; FOCUS additionally uses the finite-horizon defect penalty.

C.1 FOCUS CANDIDATE-STATE CONSTRUCTION

FOCUS constructs additional training states from a rollout produced by the current model. For an input x, let

$$s_t = (z_t, u_t),$$

where z_t is the explicit answer state and u_t is the recurrent memory. Starting from the task-defined initial state s_0^{clean}, we generate a short selection rollout

$$\tau_x = \left(s_0, s_1, \ldots, s_{K_p}\right)$$

using the current recurrent solver. This rollout is evaluated without gradient, and complete states (z_t, u_t) are stored so that replay resumes from an actually visited recurrent state rather than reconstructing either component independently.

Eligible update indices are restricted to

$$\mathcal{C}_x \subseteq \{t \in \mathcal{T} : t + h \leq K_p\},$$

which guarantees that both frontier scoring and the subsequent local h-step training rollout remain within the prespecified selection-rollout horizon. For every eligible candidate,

$$D_t = D_\theta^{(h,\rho)}(x, s_t)$$

is evaluated without gradient. FOCUS selects

$$t^\star = \arg\min_{t \in \mathcal{C}_x} |D_t - \gamma|.$$

If no candidate satisfies the task-specific eligibility conditions, the training step falls back to an update from the task-defined initial state.

Sudoku-Extreme. The explicit answer state contains digit logits

$$z_t \in \mathbb{R}^{81 \times 9}.$$

Given cells are clamped to their observed values after every recurrent update. The replay update is omitted when the short-rollout endpoint already decodes to the reference solution; otherwise the scheduled eligible indices are used. Exactness is computed with the reference target only to determine the training update and is unavailable to the inference procedure.

Maze-Hard. For Maze-Hard, the answer state contains logits indicating whether each cell belongs to the path,

$$z_t^{\text{path}} \in \mathbb{R}^{900 \times 2},$$

together with the decoder-aligned state variables used by the deterministic canonical-path decoder. We denote the complete answer state by

$$z_t = \left(z_t^{\text{path}}, z_t^{\text{dec}}\right).$$

The recurrent memory u_t is replayed together with the complete answer state.

A candidate is eligible when its deterministic decoded path is non-exact, or when it is exact at time t but becomes non-exact after the h-step continuation. The latter condition retains exact states whose subsequent recurrent evolution leaves the target solution; FOCUS then applies the same finite-horizon frontier score to all eligible states. Exactness is used only during training for candidate eligibility and is not used for inference-time search, verification, or repair.

C.2 FIXED MIX

Fixed Mix uses the same recurrent solver but replaces model-adaptive frontier selection with a predetermined mixture over training-state types. Let q_{clean}, q_{corrupt}, and q_{self} denote task-defined initial states, controlled corruptions of target states, and states visited during model rollouts. Fixed Mix samples

$$q_{\text{FM}}(s \mid x, y^\star) = \pi_{\text{clean}} q_{\text{clean}} + \pi_{\text{corrupt}} q_{\text{corrupt}} + \pi_{\text{self}} q_{\text{self}}.$$

The initial-state/corruption/rollout probabilities are $0.50/0.25/0.25$ for Sudoku and $0.45/0.35/0.20$ for Maze-Hard. It does not compute the repair-frontier score.

C.3 TASK-SPECIFIC TRAINING CONFIGURATION

Table 7 summarizes the Qwen3-1.7B FOCUS training settings for Sudoku-Extreme and Maze-Hard. We use AdamW (Loshchilov & Hutter, 2019). Fixed Mix sampling is defined in Appendix C.2; differences between the compared training schemes are described in Appendix D.2.

Table 7: Training settings for Qwen3-1.7B FOCUS on Sudoku-Extreme and Maze-Hard.

Setting	**Sudoku-Extreme**	**Maze-Hard**
Model and optimization		
Optimizer	AdamW; weight decay 0.01	AdamW; weight decay 0.01
Training updates	50,000	16,000
Effective batch size	16 (mini-batch 2, accumulation 8)	16 (mini-batch 1, accumulation 16)
Recurrent updater	Hidden size 256; 2 layers; 4 attention heads; dropout 0.05	Hidden size 256; 2 layers; 4 attention heads; dropout 0.05
Answer-update scale	0.80 (multiplier on the predicted logit increment)	0.50 (multiplier on the predicted logit increment)
LoRA	$r = 16,\ \alpha = 32$, dropout 0.05; attention and MLP projections	$r = 16,\ \alpha = 32$, dropout 0.05; attention and MLP projections
Learning-rate schedule	Linear warmup followed by half-cosine interpolation through fixed knots; recurrent-updater peak 3×10^{-4}; LoRA peak 10^{-5}	300-step warmup followed by cosine decay; recurrent-updater peak 2×10^{-4}; LoRA peak 10^{-5}
State selection and replay		
Rollout K_p; replay h	$(K_p, h) = (16, 4), (32, 4), (64, 8), (96, 8), (128, 8)$	$K_p = 16;\ h = 4$
Candidate rollout steps	$0, 2, 4, 8, 12, 16, 24, 32, 48, 64, 80, 96, 112$, restricted by $t + h \le K_p$	$0, 2, 4, 6, 8, 10, 12$
Replay schedule	Starts at update 3,000; scheduled fraction of mini-batches with auxiliary replay: $0.175 \to 0.325$; auxiliary-loss weight $w_{\text{aux}} = 0.45$	Starts at update 3,000; auxiliary-replay probability: $0.12 \to 0.30$ over 4,500 updates
Training objectives		
Defect parameters (ρ, γ, ϵ); β_{def}	$(0.985, 0, 0.05);\ 0.08$	$(0.985, 0, 0.05);\ 0.05$
Initial-state supervision	Task loss from the initialization in every mini-batch	Initial-state sampling probability 0.45 when auxiliary replay is not scheduled; overall probability at least 0.315
Task-energy coefficients	$(\lambda_{\text{CE}}, \lambda_{\text{hard}}, \lambda_{\text{c}}) = (1.00, 0.25, 0.20)$	Open-cell cross-entropy coefficient 1.00; $(\lambda_{\text{dice}}, \lambda_{\text{rank}}, \lambda_{\text{dec}}) = (0.40, 0.04, 0.10)$
Teacher supervision from the initial state	Frozen 16-step TRM; KL-loss weight 0.05; temperature 2; starts at update 800 with a 1,200-step warmup; at most 4 student steps	Cached outputs from a frozen 16-step TRM; KL-loss weight 0.005; temperature 4; starts at update 3,000 with a 2,000-step warmup; at most 4 student steps

The accompanying configuration files provide the complete learning-rate and replay schedules.

C.4 TRM TEACHER SUPERVISION

A frozen TRM teacher provides an additional temperature-scaled KL-divergence loss for training from the task-defined initial state. Teacher supervision is not applied to auxiliary replay and is not used during state selection or inference. Table 7 lists the task-specific teacher settings.

C.5 Training Algorithm

Algorithm 1 summarizes the FOCUS training procedure. The ordinary training loss samples from q_{base} (Appendix D.2) and includes teacher supervision only for initial-state training (Appendix C.4).

Algorithm 1 FOCUS training

Require: Data $\mathcal{D}$; trainable parameters $\Theta = (\theta, \phi, \psi)$; training updates U; rollout horizon K_p; replay horizon h
Ensure: Trained parameters Θ_U

for $n = 1, \ldots, U$ **do**
  $\mathcal{L} \leftarrow 0$
  for each accumulated mini-batch B_m **do**
    Sample B_m from $\mathcal{D}$; draw $a \sim \text{Bernoulli}(p_{n,m})$
    for all $(x, y^\star) \in B_m$ **do**
      Compute ℓ_{base} if Sudoku or $a = 0$; otherwise set $\ell_{\text{base}} \leftarrow 0$
      $\ell_{\text{aux}} \leftarrow 0$
      if $a = 1$ **then**
        Generate $\tau = (s_0^{\text{clean}}, \ldots, s_{K_p})$ without gradients
        Construct $\mathcal{C}_x$ as in Appendix C.1; evaluate $D_\theta^{(h,\rho)}(x, s_t)$ without gradients for $t \in \mathcal{C}_x$
        if $\mathcal{C}_x \neq \varnothing$ **then**
          $t^\star \leftarrow \arg\min_{t \in \mathcal{C}_x} \left| D_\theta^{(h,\rho)}(x, s_t) - \gamma \right|$
          $s_f \leftarrow \text{stopgrad}(s_{t^\star})$, retaining both $z_{t^\star}$ and $u_{t^\star}$
          Recompute $R_{\phi,\psi}(x)$ with gradients
          Replay h updates with gradients from s_f, using $R_{\phi,\psi}(x)$ and c_x
          Evaluate $\mathcal{L}_{\text{task}}$ and $D_\theta^{(h,\rho)}(x, s_f)$ on this replay
          $\ell_{\text{aux}} \leftarrow \mathcal{L}_{\text{task}} + \beta_{\text{def}} \left[D_\theta^{(h,\rho)}(x, s_f) - \gamma \right]_+^2$
        else
          For Maze-Hard, compute ℓ_{base}; for Sudoku, retain its existing loss
        end if
      end if
      $\mathcal{L} \leftarrow \mathcal{L} + \ell_{\text{base}} + w_{\text{aux}}^{\text{task}} \ell_{\text{aux}}$
    end for
  end for
  Update Θ using $\nabla_\Theta \left(\mathcal{L} / \sum_m |B_m| \right)$
end for

Replay probabilities, candidate steps, and loss coefficients are listed in Table 7.

D Evaluation Protocols

D.1 Source-Task Evaluation

Exact solve rate is the primary evaluation metric for Sudoku-Extreme and Maze-Hard.

Sudoku-Extreme. Each instance consists of a partially observed 9×9 Sudoku grid with a unique reference solution. Given digits are immutable: they are clamped to their observed values after every explicit-state update.

A Sudoku instance is counted as exactly solved only when the final decoded grid matches the unique reference solution in all 81 cells:

$$\text{Exact}_{\text{Sudoku}}(\hat{y}, y^\star) = \mathbb{I}\left[\hat{y}_i = y_i^\star \;\; \forall i \in \{1, \ldots, 81\}\right].$$

Exact solve rate is the mean of this indicator over the evaluation set.

For the trajectory visualizations, cell accuracy measures agreement with the reference solution over all 81 cells, including givens.

The Sudoku-Extreme corpus contains $422{,}786$ puzzles. We use its first $1{,}000$ puzzles for development and sample $1{,}000$ test puzzles uniformly without replacement from the remaining puzzles using seed

20260831. Training uses $71{,}407$ symmetry-augmented examples and $50{,}000$ synthetic puzzles. We verified that neither training source overlaps with the development or test puzzles. All compared methods use the same test set.

Maze-Hard. Maze-Hard uses fixed train/development/test splits of $900/100/1{,}000$ instances. Each maze contains a start cell S, a goal cell G, traversable cells, and walls.

A breadth-first search with fixed down–left–right–up neighbor expansion order and first-discovery parent assignment defines a unique canonical shortest-path target. This canonicalization makes exact solve well-defined even when multiple shortest paths exist geometrically.

FOCUS, Random, Energy-Hard, and Fixed Mix use a deterministic canonical-path decoder based on predicted parent probabilities and distances to the goal. Starting from the goal, the decoder repeatedly selects a traversable neighbor

$$j^\star = \arg\max_{j \in \mathcal{N}(i)} \left[\log p_{\text{parent}}(j \mid i) - 0.35\,|d_i - d_j - 1|\right],$$

where i is the current cell, $\mathcal{N}(i)$ contains its traversable neighbors, p_{parent} is the learned parent probability, and d is the learned distance prediction. Exact score ties are resolved using the same fixed down–left–right–up order.

The decoder follows at most 256 path transitions and stops earlier if it reaches the start cell or encounters a loop. An unsuccessful walk remains partial.

A Maze-Hard instance is counted as exactly solved only when the deterministically decoded path matches the protocol-defined canonical shortest-path target exactly:

$$\text{Exact}_{\text{Maze}}(\hat{y}, y^\star) = \mathbb{I}\left[\hat{y} = y^\star\right].$$

Diagnostic metrics include path validity, shortest-path optimality, and path F1. A valid path connects the start and goal through traversable cells as a single non-branching route; it is optimal if its length equals the shortest-path length. Path F1 is computed over traversable cells, excluding the two endpoints, and averaged across mazes. A valid shortest path need not match the canonical reference when multiple shortest paths exist.

The Maze One-step State-SFT and Recurrent final-only baselines instead classify path cells using their prespecified threshold; they do not require the canonical-path prediction heads. All methods are scored against the same canonical reference.

Prompt-only reasoning models. Table 2 reproduces the results reported by Wang et al. (2025) for DeepSeek-R1, Claude 3.7 8K, and o3-mini (high). All three obtain 0% exact solve accuracy on Sudoku-Extreme and Maze-Hard, meaning that no evaluated instance receives a fully correct solution. Partial grids and incomplete paths do not count as solved instances. Both tasks require coordinating decisions across the entire grid, where a locally plausible digit or path segment can conflict with the remaining solution. This motivates maintaining an explicit candidate solution that can be revised across successive updates.

Development selection. Checkpoints are selected by development exact solve rate. The state-curation comparison uses $K = 128$ on Sudoku-Extreme and $K = 16$ on Maze-Hard. For the depth analysis, K is also selected on the development set from the candidate horizons reported in the main text, with ties favoring smaller K. The test sets are used only for final evaluation.

Inference protocols. The number of recurrent updates depends on the method.

- **Direct prediction.** Direct models predict the structured output without recurrent updates. A fixed task parser or deterministic decoder produces the final answer. Malformed outputs are counted as incorrect.
- **One-step State-SFT.** One-step models initialize the task-defined state, apply exactly one learned state update, and decode once.
- **Recurrent models.** Recurrent final-only, Fixed Mix, and FOCUS models initialize the same task-defined initial state, apply exactly K updates of the shared recurrent transition, and decode once.

Each method produces one final prediction. Ground-truth targets, teacher outputs, and training-time selection scores are unavailable during prediction; reference targets are used only for scoring afterward. No additional search, verifier-guided correction, parameter optimization, or retries are used.

D.2 Baseline Definitions

We compare direct prediction, one-step state prediction, recurrent prediction, and training with additional replay states. All methods use the same task inputs and exact-match metric. Evaluation follows Appendix D.1.

Base. The unadapted pretrained or instruction-tuned backbone produces a single greedy prediction from the textual puzzle input.

Direct LoRA SFT. Only the language-model LoRA parameters are trained to predict the target directly. For Sudoku-Extreme, the output contains the 81 digits of the reference solution.

One-step State-SFT. The model applies one learned update from the task-defined initial state,

$$s_1 = F_\theta \left(s_0^{\text{clean}}; R_{\phi,\psi}(x), c_x \right),$$

and trains the resulting prediction against the reference target.

Recurrent final-only. The model repeatedly applies the shared updater from the task-defined initial state,

$$s_{t+1} = F_\theta \left(s_t; R_{\phi,\psi}(x), c_x \right), \qquad t = 0, \ldots, K-1.$$

Task supervision is applied only to the final prediction at s_K.

Fixed Mix. Additional training states are sampled from the mixture defined in Appendix C.2. This method does not use the defect penalty.

FOCUS. FOCUS uses the state-selection and replay procedure defined in Section 3 and Algorithm 1.

D.3 Defect-Penalty Ablation

We remove the defect penalty by setting $\beta_{\text{def}} = 0$, while retaining frontier selection and full-state replay. Within each task, both variants use seed 42 and the same training and evaluation settings, differing only in the defect penalty. Evaluation uses $K = 128$ on Sudoku-Extreme and $K = 16$ on Maze-Hard.

Table 8: **Defect-penalty ablation.** Qwen3-1.7B exact solve accuracy (%) with and without the penalty.

Method	Sudoku-Extreme	Maze-Hard
FOCUS w/o defect penalty	61.3	85.8
FOCUS	**64.4**	**91.1**

Both variants retain frontier selection and full-state replay. Bold marks the higher accuracy in each task.

Removing the penalty reduces exact solve accuracy from 64.4% to 61.3% on Sudoku-Extreme and from 91.1% to 85.8% on Maze-Hard (Table 8). These controlled comparisons show gains of 3.1 and 5.3 percentage points from the defect penalty in the respective configurations.

D.4 Downstream Benchmarks

We evaluate transfer to mathematical reasoning and code execution using MATH (Hendrycks et al., 2021), MATH-Hard, AIME 2025, AIME 2026, and CruxEval-O/I (Gu et al., 2024).

Mathematical reasoning. MATH contains 5,000 test problems; MATH-Hard contains the 1,324 Level-5 problems from the same test split (LightEval, 2024). AIME 2025 and AIME 2026 each contain 30 problems. We use the `test` split of `test-time-compute/aime_2025` (Test-Time Compute Organization, 2025) and the `train` split of `MathArena/aime_2026` (Dekoninck et al., 2026), respectively. Both are used exclusively for evaluation.

Code execution. CruxEval-O evaluates program-output prediction, while CruxEval-I evaluates input prediction given a program and its output. Each contains 800 examples.

Models and evaluation. We evaluate Qwen3-1.7B, Qwen3-4B, Qwen3-8B, Llama-3.2-3B-Instruct, and Llama-3.1-8B-Instruct. For each backbone, Base denotes the unadapted model; FOCUS (Sudoku) and FOCUS (Maze) use the LoRA adapters trained on Sudoku-Extreme and Maze-Hard, respectively. The recurrent updater is disabled during downstream evaluation.

Each item is evaluated with a fixed zero-shot chain-of-thought prompt and one greedy completion. Within each backbone, the compared variants use the same prompt, tokenizer, chat template, generation limit, answer extraction, and scoring procedure. Source-task checkpoints are selected using source-task development data before downstream evaluation. No downstream benchmark is used for parameter training or adapter selection, and adapters are applied with scaling coefficient one.

Mathematical answers are extracted and scored with the benchmark-specific answer verifier. CruxEval-O and CruxEval-I use their official evaluators. Malformed completions follow the same scoring rules for all variants. No retries or verifier-guided correction are used.

E RECURRENT REPAIR ON SUDOKU-EXTREME

Figure 4 shows the first 34 recurrent updates of Qwen3-1.7B FOCUS on one Sudoku-Extreme test puzzle, with the first correct solution reached at step 34.



Figure 4: Visualization of a Sudoku puzzle and its solving steps using FOCUS.

F Recurrent Repair on Maze-Hard

Figure 5 shows Qwen3-1.7B FOCUS solving a Maze-Hard test instance at step 16. Cell accuracy covers non-wall cells. The model shown achieves 90.6% accuracy at $K = 16$ and uses a different model snapshot from the one yielding 91.1% in Table 5.

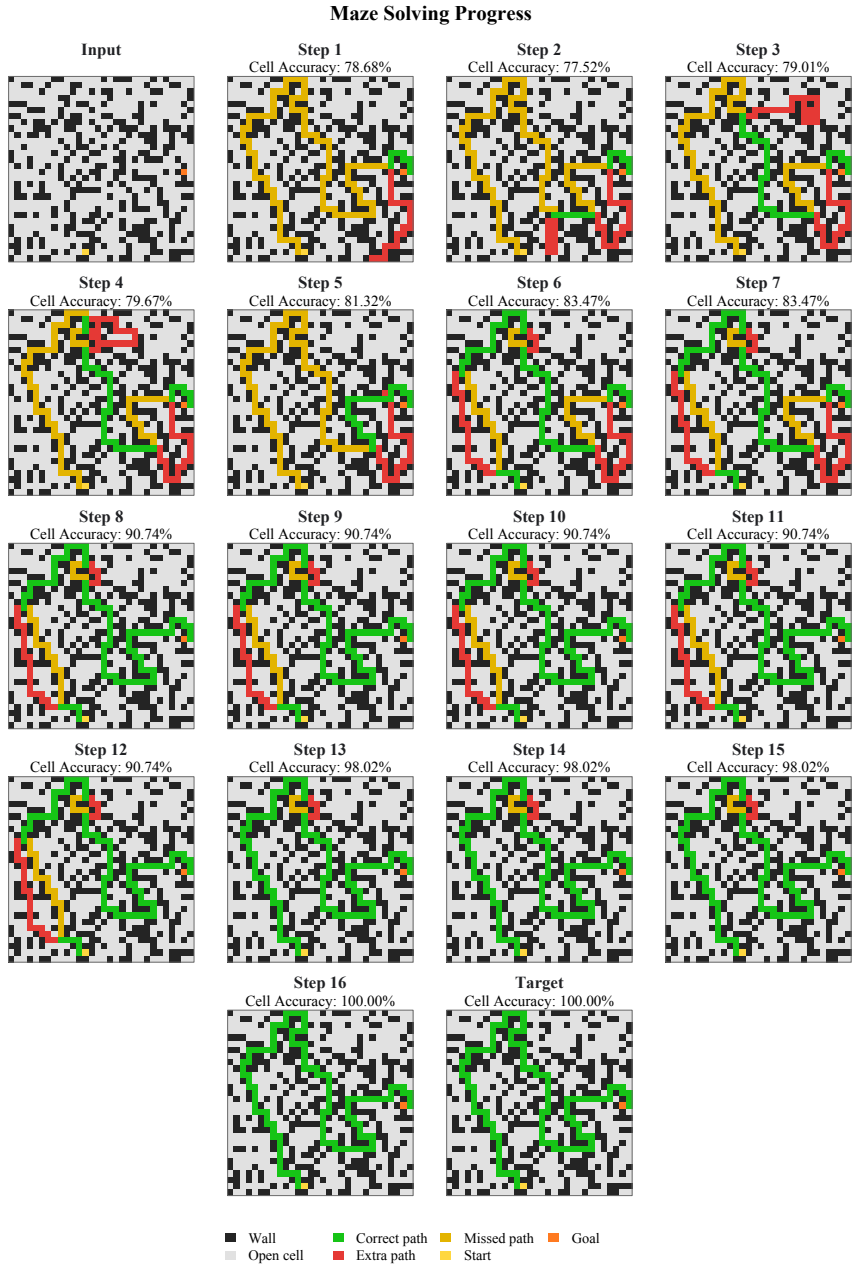

Figure 5: Visualization of a maze and its solving steps using FOCUS.

G Qualitative Comparisons with Baselines

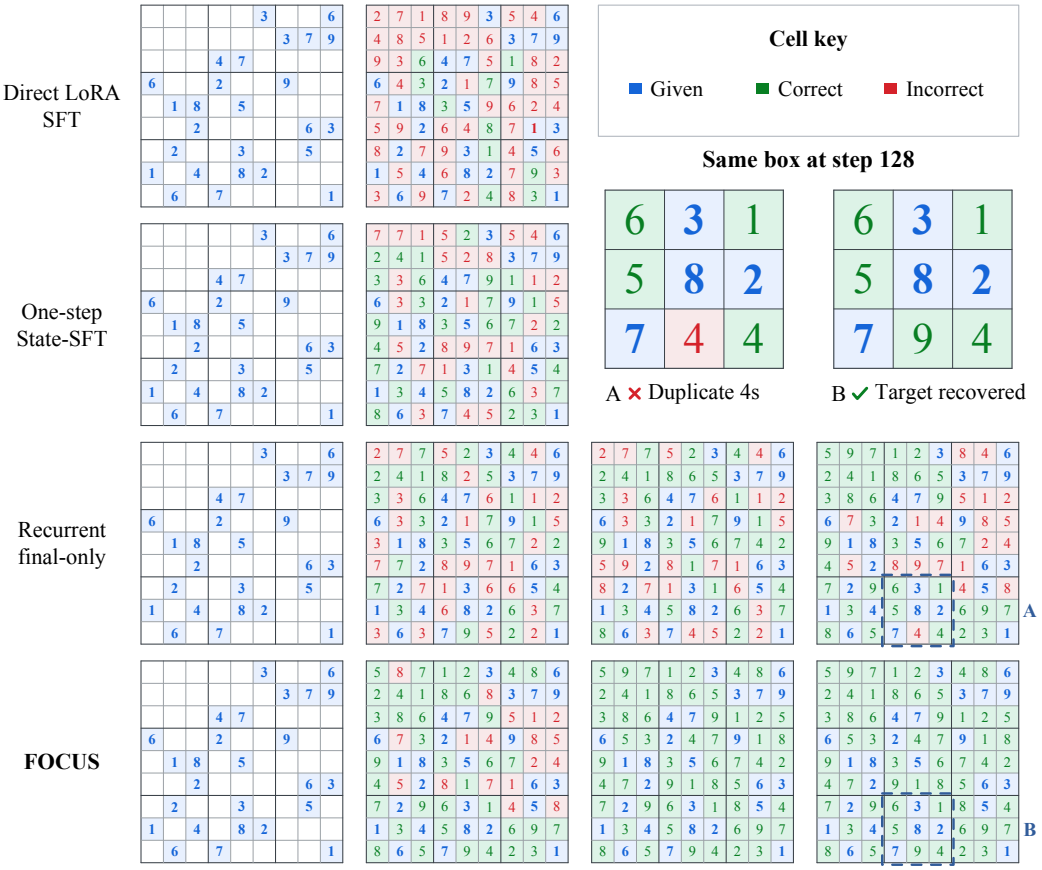

Figure 6: Predictions on the same Sudoku-Extreme test puzzle using Qwen3-8B. Each row starts with the input. Direct LoRA SFT and One-step State-SFT show their final answers in the second position; Recurrent final-only and FOCUS show predictions at steps 4, 8, and 128. FOCUS solves this example at step 8 and retains the solution through step 128. Insets A and B magnify the boxed regions at step 128. Colors indicate agreement with the target after inference.

The corresponding Maze-Hard comparison is shown in Figure 3 in the main text.

H QUALITATIVE EXAMPLES OF DOWNSTREAM TRANSFER

Prompt (MATH · Easy Question)

A line is parameterized by

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 8 \\ -1 \end{pmatrix} + t \begin{pmatrix} 2 \\ 3 \end{pmatrix}.$$

The equation of the line can be expressed in the form $y = mx + b$. Enter the ordered pair (m, b).

Baseline ✗

Response excerpts

Step 1: Eliminate the parameter t
From the x-equation:

$$t = \frac{x-8}{2}$$

From the y-equation:

$$t = \frac{y+1}{3}$$

Set the two expressions for t equal to each other:

$$\frac{x-8}{2} = \frac{y+1}{3}$$

Step 2: Solve for y in terms of x
Cross-multiply:

$$3(x-8) = 2(y+1)$$

Simplify both sides:

$$3x - 24 = 2y + 2$$

Now isolate y:

$$3x - 26 = 2y \quad \Rightarrow \quad y = \frac{3}{2}x - \frac{13}{2}$$

Step 3: Identify the slope and intercept
From the equation $y = \frac{3}{2}x - \frac{13}{2}$, we can see:
- The slope $m = \frac{3}{2}$
The y-intercept $b = -\frac{13}{2}$
Final Answer

$$\boxed{\left(\frac{3}{2}, -\frac{13}{2}\right)}$$

FOCUS ✓

Response excerpts

Step 1: Solve for t in terms of x
From the x-equation:

$$x = 8 + 2t \Rightarrow t = \frac{x-8}{2}$$

Step 2: Substitute t into the y-equation

$$y = -1 + 3t = -1 + 3\left(\frac{x-8}{2}\right)$$

Simplify:

$$y = -1 + \frac{3x-24}{2} = \frac{-2+3x-24}{2} = \frac{3x-26}{2}$$

So the equation becomes:

$$y = \frac{3}{2}x - 13$$

Step 3: Identify m and b
From the equation $y = \frac{3}{2}x - 13$, we identify:
- $m = \frac{3}{2}$
$b = -13$
Final Answer

$$\boxed{\left(\frac{3}{2}, -13\right)}$$

Figure 7: Baseline and FOCUS response excerpts on MATH (example 1).

Prompt (MATH · Hard Question)

What is the value of

$$(3x-2)(4x+1)-(3x-2)4x+1$$

when $x=4$?

Baseline ✗

Response excerpts

Step 1: Simplify the Expression
We first simplify the expression algebraically before substituting the value of x.
Let's denote:
- $A=(3x-2)(4x+1)$
$B=(3x-2)\cdot 4x+1$
So the expression becomes:

$$A-B=(3x-2)(4x+1)-[(3x-2)\cdot 4x+1]$$

[...]
Step 3: Subtract the Two Expanded Terms

$$A-B=(12x^2-5x-2)-(12x^2-8x+1)$$

Distribute the negative sign:

$$=12x^2-5x-2-12x^2+8x-1$$

Combine like terms:
- $12x^2-12x^2=0$
$-5x+8x=3x$
$-2-1=-3$
So the simplified expression is:

$$3x-3$$

Step 4: Substitute $x=4$

$$3x-3=3(4)-3=12-3=9$$

Final Answer

$$\boxed{9}$$

FOCUS ✓

Response excerpts

Step 1: Simplify the Expression
We begin by observing that the first two terms share a common factor of $(3x-2)$. So we factor that out:

$$(3x-2)(4x+1)-(3x-2)4x+1=(3x-2)[(4x+1)-4x]+1$$

Simplify the expression inside the brackets:

$$(4x+1)-4x=1$$

So the expression becomes:

$$(3x-2)(1)+1=3x-2+1=3x-1$$

Step 2: Substitute $x=4$
Now substitute $x=4$ into the simplified expression:

$$3x-1=3(4)-1=12-1=11$$

Final Answer

$$\boxed{11}$$

Figure 8: Baseline and FOCUS response excerpts on MATH (example 2).

Prompt (AIME 2026 · Easy Question)

Find the number of integers less than or equal to 100 that are equal to $a + b + ab$ for some choice of distinct positive integers a and b.

Baseline ✗

Reasoning excerpts; generation reached token limit

Alternatively, the numbers that can be written as product of two distinct integers ≥ 2 are:
- All composite numbers except the prime powers.
But also, numbers that are prime powers cannot be written as such. So, the count is total numbers from 2 to 101 minus the number of primes (since primes cannot be written as product of two integers ≥ 2) minus the number of prime powers (since they cannot be written as such).
But wait, primes are not composite, so they are excluded. So, total numbers from 2 to 101 is 100 numbers. Subtract the number of primes in that range, subtract the number of prime powers in that range. But then, the numbers that can be written as product of two distinct integers ≥ 2 are: Total numbers - primes - prime powers.
[...]
The primes between 2 and 101 are:
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97.
[...]
So, 25 primes between 2 and 101 inclusive.
Then, the number of prime powers between 2 and 101 inclusive is 10 as previously determined (4, 8, 9, 16, 25, 27, 32, 49, 64, 81). So, 10 numbers.
Therefore, the count of numbers M from 2 to 101 that can be written as product of two distinct integers ≥ 2 is 100 - 25 (primes) - 10 (prime powers) = 65.
[...]
Therefore, the answer is 65. Therefore, the number of integers less than or equal to 100 that are equal to a + b + ab for some distinct positive integers a and b is 65.
[...]
Final Answer $\boxed{65}$

FOCUS ✓

Response excerpts

Step 2: Count Valid m
We are to count the number of integers $m \leq 101$ (since $n \leq 100$) such that m can be written as the product of two distinct integers ≥ 2, i.e., $m = (a+1)(b+1)$, where $a, b \in \mathbb{Z}^+$, and $a \neq b$.
This is equivalent to counting the number of integers $m \leq 101$ that are composite and not perfect squares of primes.
Step 3: Count Composite Numbers from 6 to 101
- Total numbers from 1 to 101: 101
Number of primes up to 101: 26 (including 101)
Number of composites from 1 to 101: $101 - 26 - 1 = 74$
From 1 to 5, the only composite number is 4. So, the number of composites from 6 to 101 is:

$$74 - 1 = 73$$

Step 4: Exclude Numbers That Are Squares of Primes
We now exclude numbers m that are squares of primes and are in the range 6 to 101.
- $2^2 = 4$ (too small)
$3^2 = 9$
$5^2 = 25$
$7^2 = 49$
$11^2 = 121$ (too large)
So, the squares of primes in the range are: 9, 25, 49 $\rightarrow$ 3 numbers.
Step 5: Final Count
From the 73 composite numbers from 6 to 101, we exclude 3 that are squares of primes. Thus, the number of valid m is:

$$73 - 3 = 70$$

Each such m corresponds to a unique $n = m - 1$, and since $m \leq 101$, $n \leq 100$, the count of such n is also 70.
Final Answer

$$\boxed{70}$$

Figure 9: Baseline and FOCUS response excerpts on AIME 2026 (example 1).

Prompt (AIME 2026 · Hard Question)

A real number x satisfies $\sqrt[20]{x^{\log_{2026} x}} = 26x$. What is the number of positive divisors of the product of all possible positive values of x?

Baseline ✗

Reasoning excerpts; generation reached token limit

Let me try that again. Let x = 26^k, then log_{2026}x = k log_{2026}26 = k m, where m = log_{2026}26.
Then, the left side:

$$\sqrt[20]{x^{\log_{2026} x}} = \sqrt[20]{(26^k)^{km}} = \sqrt[20]{26^{k^2 m}} = 26^{k^2 m/20}$$

[...]
So, equation becomes:

$$26^{k^2 m/20} = 26^{k+1}$$

Which implies:

$$\frac{k^2 m}{20} = k + 1$$

Which is the same equation as before. So, k^2 m = 20k + 20.
[...]
But since m is a constant, and we need to find all positive x = 26^k, then the solutions for k are the roots of the quadratic equation:
k^2 m - 20k - 20 = 0
[...]
So, solving for k:
k = [20 ± sqrt(400 + 80m)] / (2m)
But since m is log_{2026}26, which is a positive number (since 26 < 2026), the discriminant must be positive, which it is, as 400 + 80m > 0.
But since we need positive x, k must be positive. So, we need to check if the solutions for k are positive.
[...]
Now, we need to check if k_1 and k_2 are positive.
[...]
For k_1: numerator is 20 + sqrt(...), which is positive, so k_1 is positive.
For k_2: numerator is 20 -

[Generation ends here; no final answer.]

FOCUS ✓

Reasoning excerpts; generation reached token limit

Therefore, x_1 = 2026^{y_1} and x_2 = 2026^{y_2}
So the product x_1 * x_2 = 2026^{y_1 + y_2}
But from quadratic equation, y_1 + y_2 = 20 (since for quadratic equation ay^2 + by + c = 0, sum of roots is -b/a.
Here, equation is y^2 - 20y - 20c = 0, so sum of roots is 20.
Therefore, y_1 + y_2 = 20.
Therefore, x_1 * x_2 = 2026^{20}
So the product of all possible positive values of x is 2026^{20}
[...]
To find the number of positive divisors, we need to factor 2026^{20} into its prime factors.
[...]
2026 ÷ 2 = 1013
[...]
So 1013 is a prime number. Therefore, 2026 = 2 * 1013.
Therefore, 2026^{20} = (2 * 1013)^{20} = 2^{20} * 1013^{20}
Therefore, the number of positive divisors is (20 + 1) * (20 + 1) = 21 * 21 = 441.
[...]
From the quadratic equation in y: y_1 + y_2 = 20, and x_1 = 2026^{y_1}, x_2 = 2026^{y_2}, so x_1 * x_2 = 2026^{y_1 + y_2} = 2026^{20}. Yes, that's correct.
Then, factor 2026: 2 * 1013, so 2026^{20} = 2^{20} * 1013^{20}, so number of divisors is (20 + 1)(20 + 1) = 441. So the answer is 441.
[...]
Final Answer $\boxed{441}$

Figure 10: Baseline and FOCUS response excerpts on AIME 2026 (example 2).

Prompt (MATH-Hard · Easy Question)

Find the largest prime number (in decimal form) that divides the sum,

$$1_2 + 10_2 + 100_2 + \cdots + 100000000_2.$$

Baseline ✗

Response excerpts

Step 1: Understand the Series
Each term in the series is a power of 2 in binary form:
- $1_2 = 2^0 = 1$
$10_2 = 2^1 = 2$
$100_2 = 2^2 = 4$
...
$100000000_2 = 2^9 = 512$
[...]
This is a geometric series with first term $a = 1$, common ratio $r = 2$, and 10 terms.
Step 2: Compute the Sum
The sum of a geometric series is given by:

$$S = a \cdot \frac{r^n - 1}{r - 1}$$

Here, $a = 1$, $r = 2$, and $n = 10$:

$$S = \frac{2^{10} - 1}{2 - 1} = 1024 - 1 = 1023$$

[...]
Step 3: Factorize 1023
We now find the prime factorization of 1023.
- $1023 \div 3 = 341$ (since the sum of digits is $1 + 0 + 2 + 3 = 6$, divisible by 3)
$341 \div 11 = 31$ (using the divisibility rule for 11: alternating sum of digits is $3 - 4 + 1 = 0$, divisible by 11)
So:

$$1023 = 3 \times 11 \times 31$$

[...]
Among these, the largest is:

$$\boxed{31}$$

FOCUS ✓

Response excerpts

Step 1: Understand the Series
Each term in the series is a power of 2, written in binary:
- $1_2 = 2^0 = 1$
$10_2 = 2^1 = 2$
$100_2 = 2^2 = 4$
...
$100000000_2 = 2^8 = 256$
[...]
This is a geometric series with first term $a = 1$, common ratio $r = 2$, and number of terms $n = 9$.
Step 2: Use the Formula for the Sum of a Geometric Series
The sum of a geometric series is given by:

$$S = a \cdot \frac{r^n - 1}{r - 1}$$

Substituting the values:

$$S = 1 \cdot \frac{2^9 - 1}{2 - 1} = 2^9 - 1 = 512 - 1 = 511$$

[...]
Step 3: Factor 511
We now find the prime factorization of 511.
- $511 \div 7 = 73$ (since $7 \times 73 = 511$)
73 is a prime number.
So the prime factorization of 511 is:

$$511 = 7 \times 73$$

[...]
The prime factors of 511 are 7 and 73. The largest of these is:

$$\boxed{73}$$

Figure 11: Baseline and FOCUS response excerpts on MATH-Hard (example 1).

Prompt (MATH-Hard · Hard Question)

Notice that

$$31 \cdot 37 = 1147.$$

Find some integer n with $0 \leq n < 2293$ such that

$$31n \equiv 3 \pmod{2293}.$$

Baseline ✗

Response excerpts

We first verify that 31 and 2293 are coprime. Since 31 is a prime number, we check if 31 divides 2293.

$$2293 \div 31 = 74 \text{ remainder } -1$$

[...]
So, 31 and 2293 are coprime, and the inverse of 31 modulo 2293 exists.
[...]
We apply the Extended Euclidean Algorithm to find integers x and y such that:

$$31x + 2293y = 1$$

Using the Euclidean algorithm:

$$2293 = 31 \cdot 74 - 1$$
$$31 = 30 \cdot 1 + 1$$
$$30 = 1 \cdot 30 + 0$$

Working backwards:

$$1 = 31 - 30 \cdot 1$$

Substitute $30 = 2293 - 31 \cdot 74$ into the equation:

$$1 = 31 - (2293 - 31 \cdot 74) \cdot 1$$
$$1 = 31 - 2293 + 31 \cdot 74$$
$$1 = 31 \cdot 75 - 2293$$

So, the inverse of 31 modulo 2293 is:

$$x = 75 \quad (\text{since } 31 \cdot 75 \equiv 1 \pmod{2293})$$

[...]
Multiply both sides by the inverse of 31 modulo 2293, which is 75:

$$n \equiv 3 \cdot 75 \pmod{2293}$$
$$n \equiv 225 \pmod{2293}$$

Since $0 \leq n < 2293$, the solution is:

$$\boxed{225}$$

FOCUS ✓

Response excerpts

We first verify that the modular inverse of 31 modulo 2293 exists. This is equivalent to checking whether $\gcd(31, 2293) = 1$.
Using the Euclidean algorithm:

$2293 = 31 \cdot 74 - 1 \Rightarrow \gcd(31, 2293) = \gcd(31, -1) = 1$

Since the GCD is 1, the inverse of 31 modulo 2293 exists.
[...]
We use the Extended Euclidean Algorithm to find integers x and y such that:

$$31x + 2293y = 1$$

From the earlier computation:

$$2293 = 31 \cdot 74 - 1 \Rightarrow 31 \cdot 74 = 2293 + 1$$

So:

$$31 \cdot 74 - 2293 = 1$$

This implies:

$$31 \cdot 74 \equiv 1 \pmod{2293}$$

Thus, the modular inverse of 31 modulo 2293 is:

$$31^{-1} \equiv 74 \pmod{2293}$$

[...]
Multiply both sides by the inverse of 31:

$$n \equiv 3 \cdot 74 \pmod{2293}$$
$$n \equiv 222 \pmod{2293}$$

Since $0 \leq n < 2293$, the solution is:

$$\boxed{222}$$

Figure 12: Baseline and FOCUS response excerpts on MATH-Hard (example 2).

Prompt (CruxEval-O · Easy Question)

Predict exactly what the function returns.

```
def f(text, value):
    length = len(text)
    letters = list(text)
    if value not in letters:
        value = letters[0]
    return value * length
```

Input: ’ldebgp o’, ’o’

Baseline ✗

Reasoning excerpts

First, let me look at the function step by step. The function starts by getting the length of the text. The text here is ’ldebgp o’. Let me check the length. Let’s count the characters: l, d, e, b, g, p, space, o. That’s 8 characters. So length is 8.
Then, letters is created as a list of the text. So letters would be [’l’, ’d’, ’e’, ’b’, ’g’, ’p’, ’ ’, ’o’].
Next, the function checks if value is not in letters. The value here is ’o’. So we check if ’o’ is in the letters list. Let’s see: the letters list includes ’o’ at the end. So ’o’ is in letters. Therefore, the condition is false, and the code doesn’t do anything with the value. So value remains as ’o’.
Then, the function returns value multiplied by length. So ’o’ * 8. That would be ’o’ repeated 8 times. So the result is ’ooooooo o’ (with 8 o’s). Wait, let me check: ’o’ multiplied by 8 would be ’oooooooo’ (eight o’s). Wait, the original text’s length is 8, so the output is 8 o’s. So the final answer should be ’o’*8, which is ’oooooooo’.
Wait, but let me make sure. The input text is ’ldebgp o’ which is 8 characters. The value is ’o’, which is present in the letters. So the code returns value * length, which is ’o’ * 8. So the output is ’oooooooo’ (eight o’s). So the Python repr would be ’oooooooo’.

Final response
<answer> ’oooooooo’ </answer>

Independent execution check
7 characters; expected 8.

FOCUS ✓

Reasoning excerpts

First, let me look at the function step by step. The function starts by getting the length of the text. The text here is ’ldebgp o’. Let me check the length. Let’s count the characters: l, d, e, b, g, p, space, o. Wait, that’s 8 characters? Wait, ’ldebgp o’ has 8 characters? Let me count again. ’l’ is 1, ’d’ 2, ’e’ 3, ’b’ 4, ’g’ 5, ’p’ 6, space 7, ’o’ 8. Yes, length is 8.
Then, letters is a list of the text. So letters would be [’l’, ’d’, ’e’, ’b’, ’g’, ’p’, ’ ’, ’o’].
Next, the function checks if value is not in letters. The value here is ’o’. So we check if ’o’ is in letters. Let’s see: the letters list includes ’o’ as the last element. So ’o’ is in letters. Therefore, the condition is false, so the code doesn’t do anything with the value. So value remains as ’o’.
Then, the function returns value multiplied by length. So ’o’ * 8. That would be ’o’ repeated 8 times. So the result is ’oooooooo’.
Wait, but let me make sure. The value is ’o’, which is in the letters list. So the code doesn’t change the value. So the return is ’o’ * 8. So the final answer should be ’o’ multiplied by 8, which is ’oooooooo’ (eight ’o’s). So the Python repr would be ’oooooooo’.

Final response
<answer> ’oooooooo’ </answer>

Independent execution check
8 characters; matches execution.

Figure 13: Baseline and FOCUS response excerpts on CruxEval-O (example 1).

Prompt (CruxEval-O · Hard Question)

Predict exactly what the function returns.

```
def f(array, elem):
    array.extend(elem)
    return array
```

Input: [[1, 2, 3], [1, 2], 1], [[1, 2, 3], 3, [2, 1]]

Baseline ✗

Reasoning excerpts

The input expression is: [[1, 2, 3], [1, 2], 1], [[1, 2, 3], 3, [2, 1]]. So the first argument to the function is [[1,2,3], [1,2], 1], and the second argument is [[1,2,3], 3, [2,1]]. So when the function is called, array is the first list, and elem is the second list.
So the function does array.extend(elem). So the first array is [[1,2,3], [1,2], 1], and the elem is [[1,2,3], 3, [2,1]]. So when we extend the array with elem, the new array will be the original array plus the elem. Let me think about the structure.
Original array: [[1,2,3], [1,2], 1]
Adding elem which is [[1,2,3], 3, [2,1]].
So the extended array would be:
[[1,2,3], [1,2], 1, [[1,2,3], 3, [2,1]]]
Wait, but the extend method in Python appends the elements of the iterable. So if elem is a list, then array.extend(elem) will add each element of elem to the array. So the original array is a list containing three elements: first is [1,2,3], second is [1,2], third is 1. Then elem is [[1,2,3], 3, [2,1]].
So when we extend the array with elem, the new array becomes:
[[1,2,3], [1,2], 1, [[1,2,3], 3, [2,1]]]
But wait, the elem is a list, so when you extend, each element of elem is added. So the original array has three elements, and elem has three elements. So the total length becomes 3 + 3 = 6 elements. But the question is, what is the return value of the function. The function returns the modified array.

Final response
<answer> [[1, 2, 3], [1, 2], 1, [[1, 2, 3], 3, [2, 1]]]
</answer>

FOCUS ✓

Reasoning excerpts

So the function f takes array and elem. The array is the first argument, which is a list containing three elements: [[1,2,3], [1,2], 1]. Then elem is the second argument, which is [[1,2,3], 3, [2,1]].
Wait, the function is extending the array with elem. So the array is modified by adding elem to it. So the original array is [[1,2,3], [1,2], 1], and elem is [[1,2,3], 3, [2,1]]. So when we extend the array with elem, the new array will be the original array plus elem. But wait, in Python, list.extend() adds the elements of the iterable to the list. So if elem is a list, then the array will be extended by each element of elem.
[...]
Original array: [[1,2,3], [1,2], 1]
elem is [[1,2,3], 3, [2,1]]
So after extending, the array becomes:
[[1,2,3], [1,2], 1, [1,2,3], 3, [2,1]]
Wait, but wait, the elem is a list, so the extend method adds each element of elem to the array. So the original array is [A, B, C], and elem is [D, E, F], then after extend, the array becomes [A, B, C, D, E, F]. So in this case, the original array is [[1,2,3], [1,2], 1], and elem is [[1,2,3], 3, [2,1]]. So the extended array would be the original array plus the elements of elem. So the new array is:
[...]
But wait, the function returns the array after extending. So the return value is this new array. So the final answer is the Python list representation of this new array.

Final response
<answer> [[1, 2, 3], [1, 2], 1, [1, 2, 3], 3, [2, 1]]
</answer>

Figure 14: Baseline and FOCUS response excerpts on CruxEval-O (example 2).

Prompt (CruxEval-I · Easy Question)

Provide one valid input that makes the function return the target output.

```
def f(text):
    return text.replace('\n', '\t')
```

Target output: 'apples\t\t\tpears\t\t\tbananas'

Baseline ✗

Reasoning excerpts

Okay, let's see. The function f takes a text parameter and replaces all newline characters with tabs. The target output is the string 'apples\t\t\tpears\t\t\tbananas'. So I need to find an input text that, when processed by this function, gives that output.
The function replaces '\n' with '\t'. So the target output has tabs instead of newlines. Therefore, the original text must have had newline characters that were replaced by tabs. So the input text should be something like 'apples\npears\nbananas'. Because when you replace the newlines with tabs, it becomes 'apples\t\t\tpears\t\t\tbananas'.
[...]
Wait, the replace method replaces each occurrence of the substring. So if the original text has a single newline, it becomes a single tab. But in the target output, there are three tabs between apples and pears, and three between pears and bananas. So the original text must have had three newlines, each replaced by a tab. Wait, no. Let me think again.
The target output is 'apples\t\t\tpears\t\t\tbananas'. Let's count the number of tabs. Between apples and pears, there are three tabs. So the original text must have had three newlines, each replaced by a single tab. So the original text would be 'apples\npears\nbananas'. Because each newline is replaced by a tab. So when you replace the newlines with tabs, you get the desired output.
Yes, that makes sense. So the input text is 'apples\npears\nbananas'. Then, when you run f on that, it replaces the newlines with tabs, resulting in the target output. So the answer is that input expression.

Final response
<answer> 'apples\npears\nbananas' </answer>

FOCUS ✓

Reasoning excerpts

Original text: 'apples\n\npears\n\nbananas' → when you replace each newline with a tab, the first newline becomes a tab, the second newline becomes another tab. So the text would be 'apples\t\tpears\t\tbananas' (since the two newlines are replaced by two tabs). But that's not the target output. The target output has three tabs between apples and pears, and three between pears and bananas. So maybe the original text had three newlines, each replaced by a tab. But that would make the original text have three newlines. For example, 'apples\n\n\npears\n\n\nbananas' → replacing each newline with a tab would give 'apples\t\t\tpears\t\t\tbananas', which is exactly the target output. So the original text would be 'apples\n\n\npears\n\n\nbananas'.
But the problem is to find the input text that when passed to f returns the target output. So the input text would be 'apples\n\n\npears\n\n\nbananas'. But the function replaces each newline with a tab. So the input text has three newlines, each replaced by a tab. So the output is the target.
Alternatively, maybe the input text has two newlines, but each replaced by three tabs. But that's not possible because each newline is replaced by a single tab. So the number of tabs in the output is equal to the number of newlines in the input. So the target output has three tabs between apples and pears, which implies that the input had three newlines. So the input text would be 'apples\n\n\npears\n\n\nbananas'.
[...]
But let me check. Let's take the input as 'apples\n\n\npears\n\n\nbananas'. Then, when we replace each newline with a tab, the result is 'apples\t\t\tpears\t\t\tbananas', which matches the target output. So that's the correct input.

Final response
<answer> 'apples\n\n\npears\n\n\nbananas' </answer>

Figure 15: Baseline and FOCUS response excerpts on CruxEval-I (example 1).

Prompt (CruxEval-I · Hard Question)

Provide one valid input that makes the function return the target output.

```
def f(arr):
    result = []
    for item in arr:
        try:
            if item.isnumeric():
                result.append(int(item)*2)
        except ValueError:
            result.append(item[::-1])
    return result
```

Target output: [182, 32]

Baseline ✗

Reasoning excerpts

Okay, let's see. I need to find an input array that when passed to the function f will produce [182, 32]. Let me understand how the function works.
[...]
So the target output is [182, 32]. Let's break down each element.
First element is 182. How can that come from the function? Let's think. The first element must be a numeric value. Let's see: if the item is a number, then int(item)*2 would be 182. So 182 / 2 = 91. So the original item would be 91. But wait, the item is numeric, so it's a number. So the first element in the input array must be 91. Then, 91 * 2 is 182. So the first element of the input array is 91.
Second element is 32. Let's see. The second element in the output is 32. So either the item was numeric and multiplied by 2, or it was not numeric and reversed. Let's check both possibilities.
If it's numeric: 32 / 2 = 16. So the original item would be 16. Then 16 * 2 = 32. So the second element of the input array is 16.
[...]
Wait, so the second element in the output is 32. So if the input item is 16, then 16 * 2 is 32. So the input array's second element is 16. Then, the output is 32. So the input array would be [91, 16]. Let me check that.
First element: 91 is numeric, 91*2=182. Correct.
Second element: 16 is numeric, 16*2=32. Correct. So the input array is [91, 16]. So the answer would be f([91, 16]) returns [182, 32].

Final response
<answer>[91, 16]</answer>

Independent execution check
Integer items have no `isnumeric()` method: this raises `AttributeError`, not the caught `ValueError`. Type requirement violated.

FOCUS ✓

Reasoning excerpts

Okay, let's see. I need to find an input array that when passed to the function f will produce the output [182, 32]. Let me understand how the function works.
[...]
First element is 182. Let's think: 182 is even, so maybe the original item was 91, because 91*2 is 182. But wait, the function checks if the item is numeric. So if the item is numeric, it's converted to int and multiplied by 2. So if the original item was 91, then it would be 182. But if the item was a string like "91", then it would be converted to 91, multiplied by 2, which is 182. So that's possible.
[...]
But what if the array is ["91", "16"]? Then the first item is "91" (numeric), so 91*2=182. The second item is "16" (numeric), 16*2=32. So the output is [182, 32], which matches the target.
[...]
Yes. So the function would process each item:
First item: "91" is numeric, so int("91") is 91, multiplied by 2 is 182. Then the second item is "16", which is numeric, 16*2=32. So the output is [182, 32], which is correct.
So the input array is ["91", "16"].
[...]
But let me check again. Let's simulate the code with that input.
arr = ["91", "16"]
[...]
First item is "91": isnumeric() is True. So int("91") is 91. 91 * 2 = 182. So append 182.
Second item is "16": isnumeric() is True. 16*2=32. Append 32.
So result is [182, 32], which matches the target.

Final response
<answer>["91", "16"]</answer>

Independent execution check
String items pass `isnumeric()`; conversion and doubling produce `[182, 32]`. Valid input; target reproduced.

Figure 16: Baseline and FOCUS response excerpts on CruxEval-I (example 2).