# Interference Search

Reasoning over merged states, many branches at once

Al-ameen, Bad Theory Labs
Lagos, September 2026

**Abstract.** Language models reason in a line: one token after another in a single transcript, rewinding in text when a step fails. I study a different shape. Interference Search works on explicit states. Every live branch expands at once, the environment executes the moves, branches that reach the same state merge, a small trained judge cancels dead ends, and the survivors advance together one level at a time. The name comes from the way quantum search lets wrong paths cancel; the method is classical and claims no quantum speedup. On 30 hard Countdown problems, with the same judge and the same budget of 200 judged positions, Interference Search solves 30 and a single line of thought solves 21; to solve all 30, the line needs 23.7 sequential steps on average and the frontier needs 3. Most reasoning in this domain is duplicate work: at six numbers, 831,176 operation sequences collapse into 13,229 distinct states. The judge, trained on four- and five-number problems, cut the search 12.6 times on seven-number problems without losing a solution, but loses solutions on hard problems when set aggressively. Qwen3-1.7B thinking in text solves 3 of the 30 problems and generates about 14,900 tokens per solved problem, where the search generates none; in one recorded trace the model wrote the correct expression at token 1,313 and never committed to it. On 30 MBPP problems the model fails on its first try, Interference Search solves 9 against 8 for independent sampling and 7 for both linear refinement strategies, a lead within noise. Several obvious alternatives failed, and I report them: textual notes about failed attempts made the model retry them, decode-time rewinding regenerated the same mistake, and parallel streams that could attend to each other learned nothing that independent streams did not. None of these results trains the language model; that is the next step.

## 1 The problem with a line

A reasoning model today holds one line of thought. It can explore alternatives only by writing them out in sequence, and the alternatives it has dropped stay buried in the transcript. Running several copies in parallel does not fix this: independent samples from one model share its biases, so they tend to fail in the same way. In my Countdown runs with Qwen3-1.7B [22], 45.1% of failed attempts repeated an attempt the same sample had already rejected, and a further 29.7% repeated one that a sibling sample had rejected earlier.

The clearest case is a single trace. Given $[24, 98, 19, 3]$ and the target 361, the model wrote $((24\times19)-98)+3 = 361$ at token 1,313. It checked the expression nine more times, drifted back into wrong combinations, and reached its 2,048-token budget without giving an answer. The model could find the answer. A single transcript gave it no way to set that answer beside the alternatives it was still exploring and commit to it.

Quantum search offers a picture of the alternative. Grover's algorithm [7] applies one check to a superposition of every candidate and uses the sign of the amplitudes so that wrong candidates fade together. A classical machine cannot reproduce the speedup, and I do not claim one. Two parts of the picture do survive classically. Many paths collapse into far fewer states when the problem has structure, so a reasoner that works on states handles many paths at the cost of one state each. And one refutation of a state removes every path through it, including paths that were never written out.

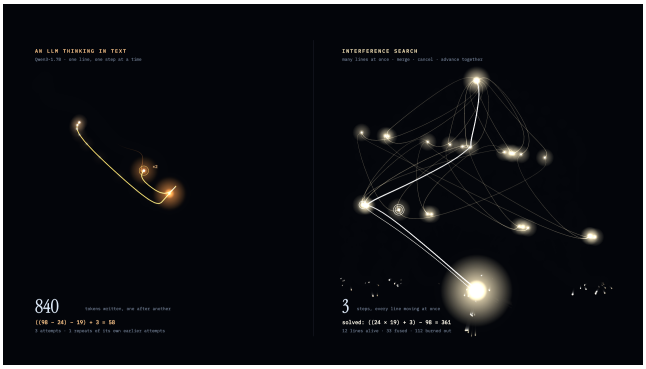


Figure 1: The same problem solved two ways. Left: Qwen3-1.7B's recorded reasoning, with every attempt placed where it was written; the rings mark repeats. Right: Interference Search expanding many branches, merging duplicates, dropping dead ends and reaching $((24 \times 19) + 3) - 98 = 361$ in three steps.

**Contributions.**

- Interference Search, a search loop over explicit states with merging, a learned judge and a level-synchronous frontier, applied to both thinking (proposing and judging) and execution (applying moves, running tests) (§2).
- Exact measurements of how much of Countdown's search space is duplicate or dead, and a judge that generalizes to larger problems than it was trained on (§3, §4).
- Matched-compute comparisons and ablations showing that the gain comes from merging plus level-synchronous advance, and that neither a global ranked memory nor plain cross-stream attention reproduces it (§5).
- Results with a real language model and in code, token accounting for both, and a set of negative results that constrain where the mechanism has to live (§6–§8).

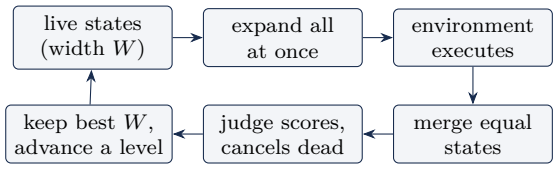


Figure 2: One round of Interference Search. Nothing carries between rounds except the states and a memory of merge keys already expanded.

## 2 Interference Search

A domain supplies six functions: a start state, a proposer that returns candidate next states for each live state, a merge key that says when two states are the same situation, a judge that scores states, a goal test and a dead-end test. Cost is counted in the domain's own unit: judged states in Countdown, generated tokens in code.

Each round (Figure 2) expands every live state, executes the proposed moves, and pools the resulting states by merge key. A state that several parents reach is kept once; the ranks at which parents proposed it are pooled as votes. States whose keys were already expanded are dropped, which is the memory that stops a branch from revisiting its own failures. The judge scores the pool, the best $W$ survive, and all survivors move to the next level together. The width $W$ is set from the budget, so extra budget widens the frontier while its depth stays fixed by the problem. The loop never carries a transcript; each proposal sees one state.

The baseline for every comparison uses the same six functions in a single chain: judge the children of the current state, sample one in proportion to its score, continue, and restart from the beginning when the chain dies.

## 3 Where the redundancy is

Countdown makes the structure exact. A state is the sorted multiset of available numbers; a move combines two of them with $+$, $-$, $\times$ or $\div$, with positive whole results; the goal is a single number equal to the target. Dynamic programming over states labels every state as alive or dead.

The arithmetic in each move is easy, and in the search arms the environment performs it; the model never computes a result. What makes Countdown hard is choosing which two numbers to combine, with which operation, in what order. That choice is the search this paper is about, and it is why Stream of Search [6] and APR [11] use the same puzzle.

Table 1 shows the two facts the method rests on. Paths per state grow by 3.2 and then 5.5 times as the fifth and sixth numbers are added, so the saving from working on states grows faster than the problem does. And after only two moves, more than 80% of all complete paths already run through a dead state; a correct refutation there removes most of the search in one step.

With the language model, the redundancy is visible directly. Across eight independent samples on six hard problems (381 failed attempts), 29.7% of failures repeated a failure a sibling had produced earlier in token time, costing 16.5% of all generated tokens on my preregistered metric, and 45.1% repeated the sample's own earlier failure. Another 46% of tokens came after some sample had already found an answer; that stopping signal is what consensus-based early termination such as Parallel-Probe [27] exploits, and I do not count it here.

| Numbers | Paths | States | Paths/state | Dead share |
|---|---|---|---|---|
| 4 | 566 | 159 | 3.6× | 95.4% |
| 5 | 17,266 | 1,511 | 11.4× | 96.2% |
| 6 | 831,176 | 13,229 | 62.8× | 82.6% |

Table 1: Path compression at the deepest non-terminal level, mean over 20 random solvable instances per size. Dead share: fraction of all complete paths that pass through a state already dead after two moves.

| Test set | Pruning | Solved | Work |
|---|---|---|---|
| 6 numbers, random (100 problems) | none | 100% | 100% |
| | hand bound | 100% | 97.8% |
| | judge, $\tau = 0.7$ | 100% | 9.3% |
| | judge, $\tau = 0.9$ | 86% | 3.7% |
| | exact oracle | 100% | 1.1% |
| 6 numbers, hard ($\leq$24 solution paths) | judge, $\tau = 0.2$ | 98% | 17.9% |
| | judge, $\tau = 0.5$ | 78% | 7.3% |
| | judge, $\tau = 0.7$ | 43% | 4.3% |
| 7 numbers, random (10 problems) | judge, $\tau = 0.7$ | 100% | 7.9% |
| | judge, $\tau = 0.9$ | 90% | 3.3% |

Table 2: Merged-state breadth-first search that keeps only states the judge scores at or above $\tau$. Work: states expanded relative to no pruning. The judge never saw six- or seven-number problems in training. The hand bound prunes when even the largest reachable value is below the target.

## 4 A judge that generalizes

The Countdown judge is a two-layer set transformer (width 64) over one feature vector per number, describing each number relative to the target: log size, distance, divisibility and residues modulo 2 to 12. It is permutation-invariant and accepts any count of numbers. I trained it on all 632,279 reachable states of 400 random four-number and 400 five-number problems, labelled by exact DP (2.6% alive), for six epochs.

Table 2 tests it by pruning, on sizes it never saw. On random six-number problems it removes 90.7% of the search without losing a solution, where a sound hand bound removes 2.2%. Two sizes beyond training it removes 92.1%, on a small set. Random problems flatter the judge, because they often have many solutions and pruning a few still leaves one alive. On hard problems with at most 24 solution paths, the setting that was safe on random problems loses more than half of them; the safe setting there is $\tau = 0.2$, which still cuts the search

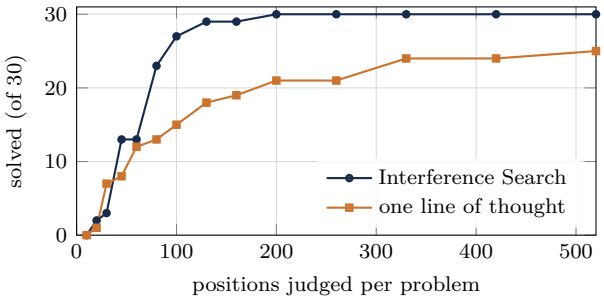


Figure 3: Hard four-number Countdown, same judge and budget for both. The frontier reaches 30 of 30 at 200 judged positions; the line reaches 25 with more than twice that.

| | One line | | Interference Search | |
|---|---|---|---|---|
| Budget | Solved | Steps | Solved | Steps |
| 150 | 19 | 6.8 | 29 | 3.0 |
| 200 | 21 | 7.9 | 30 | 3.0 |
| 500 | 24 | 10.8 | 30 | 3.0 |
| 1,500 | 30 | 23.7 | 30 | 3.0 |

Table 3: Hard four-number problems (30). Steps: mean sequential rounds on solved problems.

5.6 times at a 2% loss.

## 5 The shape of the search

Figure 3 and Table 3 compare the two shapes with the judge held fixed. The frontier solves every problem by 200 judged positions. The line needs 1,500 to solve them all, and then takes 23.7 sequential steps on average against the frontier's 3. Because branches advance together, time follows the depth of the problem rather than the number of mistakes. Wall-clock time on a laptop CPU is a few milliseconds per problem for both (8 to 14 ms for the frontier, 11 to 24 ms for the line) because my implementation executes branches one after another; the step count is what translates into latency when each step is an expensive model call executed in parallel.

To locate the gain I compared four arms on 100 unseen six-number problems at matched expansions (Figure 4). At 100 expansions the merged frontier solves 77%, the line 40%, and the frontier without merging 50%: duplicate states otherwise fill the width, and merging accounts for most of the compute advantage. A global ranked memory of every state seen (best-first search) solves 10%. It keeps choosing shallow states that look safe and never commits to depth, which is why the frontier's level-synchronous advance matters. The frontier finishes in 5 sequential rounds against the line's 116. On seven-number problems the line catches up at large budgets (87% for both at 400 expansions); the frontier's advantage is at small and medium budgets.

A separate experiment asked whether parallel streams discover coordination on their own. In a hidden-goal tree search where every stream sees the same noisy hints,

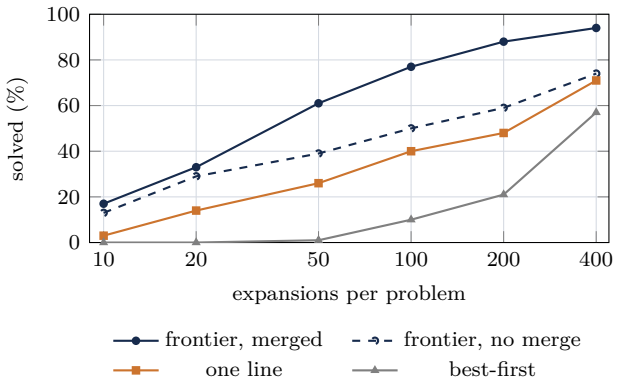


Figure 4: Unseen six-number problems (100), same judge for every arm. Merging carries most of the frontier's advantage; a global best-first memory does worse than a single line.

| System | AUC | Solved |
|---|---|---|
| Qwen3-1.7B thinking in text | | 3 |
| Interference Search, judge = asking the model yes or no | 0.58 | 4 |
| Interference Search, judge = linear probe on the model's hidden state | 0.87 | 15 |
| Interference Search, judge = linear probe on number features | 0.94 | 22 |
| Interference Search, judge = trained set transformer | 0.98 | 23 |

Table 4: Hard four-number problems (30). AUC: separating alive from dead states on unseen four-number problems. Probes were trained on 1,376 states from 60 other problems; the trained judge on 632,279.

I trained small transformer policies from scratch with grouped policy gradients, with streams either isolated or able to attend to each other. The two tied (0.508 and 0.511 success with four streams), and both stayed below a simple stochastic player with its own memory (0.70) and well below one with shared memory (0.82). Visibility alone did not produce sharing. Coordination had to be an explicit operation, which is what merging and the judge provide.

## 6 With a language model

On the same 30 hard problems, Qwen3-1.7B with thinking enabled and a 1,500-token budget solves 3. Table 4 places it against the search, where the environment lists moves and a judge ranks states.

Two readings follow. The structure is doing the work: even a probe on crude features, inside the frontier, solves seven times as many problems as the model reasoning alone, in 3 rounds instead of about 1,400 generated tokens. And the model's hidden state carries reachability information that its answer does not (0.87 against 0.58), but no more than simple number features do. I had first read the probe result as the model knowing more than it can say; the feature control does not support that reading.

| Strategy (1,500 generated tokens) | Solved |
|---|---|
| Agent loop with full chat history | 7 / 30 |
| Revise the latest program | 7 / 30 |
| Fresh attempts (best of $N$) | 8 / 30 |
| Interference Search | 9 / 30 |

Table 5: MBPP problems [4] that Qwen3-1.7B fails on its first greedy attempt (30 of 69 such problems among the first 150). A problem counts as solved only within the budget.

Three attempts to get this behaviour out of the model without training failed. *Textual ledgers:* every 256 tokens, eight streams were told which full expressions they or their siblings had already refuted. Within 80 tokens of a note, 33.6% of attempts repeated a sibling's refutation, against about 21% without notes; telling the model an expression is wrong primes it. Solve rates did not change (22, 21 and 22 of 24). *Decode-time rewinding:* when a stream wrote an already-refuted expression it was rewound and resampled; it regenerated the same line up to 16 times, because the cause sits in the context. *State prompting:* showing the model only the current state and asking for a move made it add the first two listed numbers and repeat the same three-step path; with memory and shuffled order it still solved none of a small set.

# 7 Code

In code the proposer has to invent the options. The state is a program together with its behaviour: what it returns or raises on each test. The interpreter is the executor; programs with identical behaviour merge; the judge is the number of tests passed. Each proposal sees only the task, the current program and its test results.

Table 5 shows parallel strategies ahead of both linear ones, and Interference Search one problem ahead of independent sampling, which is within noise on 30 problems. Behaviour merging removed 83% of generated programs as duplicates. The agent loop solved nothing after its third round while its history grew. The traces explain the ceiling: when this untrained model's idea is wrong, its revisions reproduce it, so here the search organizes the model's ideas without adding new ones. A model trained inside the frontier is rewarded for distinct live branches, since duplicates merge and earn nothing, and read-only actions such as retrieval can branch freely; both are ways past this ceiling, and both are untested here.

# 8 Token consumption

Table 6 counts language-model tokens. In Countdown, chain of thought generated about 14,900 tokens per solved problem (44,703 over 3 solves). Interference Search generates none: the environment writes the moves and the judge only reads each state, once, in a single forward pass. With the judge read from Qwen3-1.7B's hidden state, the search read about 8,600 tokens per solved problem. Only 18 of each 66-token judge prompt are specific to the state, so with the shared instruction prefix cached, the new tokens fall to about 1,200 per problem. Reading is also the cheaper operation: prompt tokens are processed in parallel, and hosted models usually price them below generated tokens. The two Countdown systems are not the same model doing different work, since the search runs the environment and a judge; the comparison says what each approach spends to solve these problems.

| *Countdown, 30 hard problems* | | | |
|---|---|---|---|
| System | Written | Read | Solved |
| Qwen3-1.7B thinking in text | 1,490 | 83 | 3 |
| Interference Search, hidden-state probe judge | 0 | 4,290 | 15 |
| Interference Search, trained judge | 0 | 0 | 23 |
| *Code, 30 MBPP first-try failures* | | | |
| Strategy | Written | Read | Solved |
| Agent loop with full history | 1,218 | n/r | 7 |
| Revise the latest program | 1,273 | 4,216 | 7 |
| Fresh attempts (best of $N$) | 1,357 | 1,473 | 8 |
| Interference Search | 1,368 | 1,934 | 9 |

Table 6: Language-model tokens per problem, averaged over all 30 problems. Written: generated tokens. Read: prompt tokens processed. The Countdown judge arms read one 66-token prompt per judged state, about 65 states per problem, and generate nothing. n/r: not recorded; that arm re-reads its growing history every round.

In code the comparison holds the model and the budget fixed. Every strategy spends close to its 1,500-token allowance on the problems it fails, so the per-problem averages are similar by design, and the fair measure is tokens per solved problem. Interference Search generated the fewest, about 4,560 per solve against 5,090 for fresh attempts and 5,460 for revising the latest program, and per solved problem read 2.8 times fewer tokens than the revision strategy (6,450 against 18,070). It read somewhat more than fresh attempts, because revising a program means showing it the program. The margins are modest and track the solve counts in Table 5.

# 9 Related work

**Search over reasoning.** Tree of Thoughts [24], Graph of Thoughts [5] and RAP [8] organize model outputs as search over thoughts; self-consistency [20] samples in parallel and votes over chains of thought [21]. FETCH [19] identifies over-exploration of semantically equivalent states as a failure of tree search and merges them by embedding clustering. Transposition tables [29] and learned value functions in AlphaZero [14] are the classical roots of merging and judging.

**Parallel reasoning in one model.** APR [11] trains spawn and join operations and reports large gains over serial search on Countdown; ThreadWeaver [10], Parallel-R1 [26], Hogwild! Inference [13] and Multi-Stream LLMs [15] let threads run concurrently or see each other. Parallel-Probe [27] and DeepPrune [18] prune redundant branches from outside. Parason [25] finds that most parallelizable steps in reasoning traces are competing trials. None of these merges the states branches reach or learns to cancel them inside a synchronized frontier.

**State instead of transcript.** Atom of Thoughts [17], the Markovian Thinker [1] and PENCIL [23] reduce or bound what a model carries between steps. Continuous thoughts can hold several search frontiers in theory [28], though fine-tuned models appear not to use this [12].

**Learned search and execution.** Stream of Search [6] and Searchformer [9] train models on search traces; learned value functions generalize to larger planning problems [2]. SWE-Search [3] and ParallelEnv [16] branch software agents over environment states.

## 10 Limitations

Every result is one seed, and the larger claims rest on 30 problems per condition, 10 in the seven-number test. The Countdown problems are small enough that a frontier model would likely solve most of them by reasoning in text; the comparisons here hold the model and judge fixed and vary only the shape of the search, and I have not run them at frontier scale. The strongest Countdown numbers use a small trained judge and an environment that lists moves; they do not run the language model, and the 5 to 14 ms per problem they take is not a speedup of the model itself. The judge exists only for Countdown, where states are exact and labels come from a solver; open domains need learned merge keys and labels from rollouts, which I have not built. In code the gain over independent sampling is within noise. Nothing here trains the language model, so whether a model trained to reason this way keeps these gains is open. Budgets are set in the domain's unit (judged states or generated tokens). Prompt tokens differ between arms; Table 6 reports them separately, and they are not part of any budget.

## 11 Conclusion

Holding the model and the judge fixed and changing only the shape of the search, from one chain over a transcript to a frontier over merged states, turned 21 solved problems into 30 at equal compute and 24 sequential steps into 3. The gain comes from merging and level-synchronous advance together, and it grows with problem size because the number of paths grows much faster than the number of states. The negative results say where the mechanism must live: prompts and decoding tricks do not produce it in a small model. The next step is to train it into one.

**Code and data.** https://github.com/Badtheorylabs/interference-search, under Apache 2.0, with the trained judge, every raw result and the research log. `paper/CLAIMS.md` maps each number in this paper to the file and command that produced it.

## References

[1] Milad Aghajohari, Kamran Chitsaz, Amirhossein Kazemnejad, Sarath Chandar, Alessandro Sordoni, Aaron Courville, and Siva Reddy. The Markovian Thinker: Architecture-Agnostic Linear Scaling of Reasoning. arXiv:2510.06557, 2025.

[2] Michael Aichmüller, Yannik Hesse, and Hector Geffner. Learning to Search and Searching to Learn for Generalization in Planning. arXiv:2605.25720, 2026.

[3] Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. SWE-Search: Enhancing Software Agents with Monte Carlo Tree Search and Iterative Refinement. arXiv:2410.20285, 2024.

[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program Synthesis with Large Language Models. arXiv:2108.07732, 2021.

[5] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. Graph of Thoughts: Solving Elaborate Problems with Large Language Models. arXiv:2308.09687, 2023.

[6] Kanishk Gandhi, Denise Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah D. Goodman. Stream of Search (SoS): Learning to Search in Language. arXiv:2404.03683, 2024.

[7] Lov K. Grover. A fast quantum mechanical algorithm for database search. arXiv:quant-ph/9605043, 1996.

[8] Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. Reasoning with Language Model is Planning with World Model. arXiv:2305.14992, 2023.

[9] Lucas Lehnert, Sainbayar Sukhbaatar, DiJia Su, Qinqing Zheng, Paul Mcvay, Michael Rabbat, and Yuandong Tian. Beyond A*: Better Planning with Transformers via Search Dynamics Bootstrapping. arXiv:2402.14083, 2024.

[10] Long Lian, Sida Wang, Felix Juefei-Xu, Tsu-Jui Fu, Xiuyu Li, Adam Yala, Trevor Darrell, Alane Suhr, Yuandong Tian, and Xi Victoria Lin. ThreadWeaver: Adaptive Threading for Efficient Parallel Reasoning in Language Models. arXiv:2512.07843, 2025.

[11] Jiayi Pan, Xiuyu Li, Long Lian, Charlie Snell, Yifei Zhou, Adam Yala, Trevor Darrell, Kurt Keutzer, and Alane Suhr. Learning Adaptive Parallel Reasoning with Language Models. arXiv:2504.15466, 2025.

[12] Michael Rizvi-Martel, Guillaume Rabusseau, and Marius Mosbach. The Illusion of Superposition? A Principled Analysis of Latent Thinking in Language Models. arXiv:2604.06374, 2026.

[13] Gleb Rodionov, Roman Garipov, Alina Shutova, George Yakushev, Erik Schultheis, Vage Egiazarian, Anton Sinitsin, Denis Kuznedelev, and Dan Alistarh. Hogwild! Inference: Parallel LLM Generation via Concurrent Attention. arXiv:2504.06261, 2025.

[14] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[15] Guinan Su, Yanwu Yang, Xueyan Li, and Jonas Geiping. Multi-Stream LLMs: Unblocking Language Models with Parallel Streams of Thoughts, Inputs and Outputs. arXiv:2605.12460, 2026.

[16] Shangyin Tan, Jialin Zhang, and Matei Zaharia. Parallel Environments for Agents. Demonstration, ACM Conference on AI and Agentic Systems (CAIS), 2026.

[17] Fengwei Teng, Quan Shi, Zhaoyang Yu, Jiayi Zhang, Yuyu Luo, Chenglin Wu, and Zhijiang Guo. Atom of Thoughts for Markov LLM Test-Time Scaling. arXiv:2502.12018, 2025.

[18] Shangqing Tu, Yaxuan Li, Yushi Bai, Lei Hou, and Juanzi Li. DeepPrune: Parallel Scaling without Inter-trace Redundancy. arXiv:2510.08483, 2025.

[19] Ante Wang, Linfeng Song, Ye Tian, Dian Yu, Haitao Mi, Xiangyu Duan, Zhaopeng Tu, Jinsong Su, and Dong Yu. Don't Get Lost in the Trees: Streamlining LLM Reasoning by Overcoming Tree Search Exploration Pitfalls. arXiv:2502.11183, 2025.

[20] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-Consistency Improves Chain of Thought Reasoning in Language Models. arXiv:2203.11171, 2022.

[21] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903, 2022.

[22] An Yang, Anfeng Li, Baosong Yang, et al. Qwen3 Technical Report. arXiv:2505.09388, 2025.

[23] Chenxiao Yang, Nathan Srebro, David McAllester, and Zhiyuan Li. PENCIL: Long Thoughts with Short Memory. arXiv:2503.14337, 2025.

[24] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. arXiv:2305.10601, 2023.

[25] Zhengyang Zhang, Zijian Zhang, Jiaxuan Gao, Shusheng Xu, Yi Wu, Song Han, and Ligeng Zhu. Parason: Revealing Subtask and Trial Parallelism in LLM Reasoning. arXiv:2608.24658, 2026.

[26] Tong Zheng, Hongming Zhang, Wenhao Yu, Xiaoyang Wang, Runpeng Dai, Rui Liu, Huiwen Bao, Chengsong Huang, Heng Huang, and Dong Yu. Parallel-R1: Towards Parallel Thinking via Reinforcement Learning. arXiv:2509.07980, 2025.

[27] Tong Zheng, Chengsong Huang, Runpeng Dai, Yun He, Rui Liu, Xin Ni, Huiwen Bao, Kaishen Wang, Hongtu Zhu, Jiaxin Huang, Furong Huang, and Heng Huang. Parallel-Probe: Towards Efficient Parallel Thinking via 2D Probing. arXiv:2602.03845, 2026.

[28] Hanlin Zhu, Shibo Hao, Zhiting Hu, Jiantao Jiao, Stuart Russell, and Yuandong Tian. Reasoning by Superposition: A Theoretical Perspective on Chain of Continuous Thought. arXiv:2505.12514, 2025.

[29] Albert L. Zobrist. A New Hashing Method with Application for Game Playing. Technical Report 88, University of Wisconsin, 1970.