God's Algorithm for Generalized Rubik's Cubes: A Polynomial Optimal Solver and Modular Search Framework

Shyamal Suhana Chandra
Sapana Micro Software
sapanamicrosoftware@gmail.com

June 3, 2026

Abstract

We present an open-source implementation of the polynomial-time optimal solver for $c_1 \times c_2 \times n$ Rubik's cubes described by Demaine et al. in Section 6 of *Algorithms for Solving Rubik's Cubes* [2], together with a modular search framework exposing more than thirty graph-search techniques through a unified command-line interface. The core solver decomposes the puzzle into long configurations, cubie clusters, and precomputed short-move tables, then applies dynamic programming or a long-move tour fallback. Around this backbone we integrate classical methods (BFS, DFS, A*, IDA*, bidirectional search), structured polytree and abstract space search, distributed-deferred (DD) parallel frontiers, Monte Carlo tree search, Kohonen self-organizing map (SOM) guidance [6], linear cube factorization with stochastic gradient descent and backpropagation, non-negative matrix factorization (NMF) of cubee positional-gram features [8, 9], and tree-depth reset (TDR) bidirectional zippers. A live ncurses terminal UI visualizes the flattened cube net, search progress, and optional MIDI countdown. Benchmarks on $3 \times 3 \times 3$ and $4 \times 4 \times 4$ instances (five random scrambles each, 500k node cap) show that bidirectional and zipper methods scale most reliably, while table-heavy Demaine solvers and exhaustive search degrade under fixed time and memory budgets as n grows. We summarize architecture, empirical findings, limitations, and directions for future work.

1 Introduction

The Rubik's cube and its generalizations form a canonical benchmark for combinatorial search, group theory, and algorithm engineering. For the standard $3 \times 3 \times 3$ cube, God's number—the worst-case shortest solution length—is 20 moves in the quarter-turn metric. Generalized "brick" puzzles with rectangular short faces $c_1 \times c_2$ and long axis length n admit a richer structure: short moves act along the long axis within cubie clusters, while long moves rearrange entire stacks.

Demaine et al. proved that for *fixed* c_1 and c_2, an optimal solution can be found in time polynomial in n [2, 3]. This project implements that Section 6 algorithm and extends it with a research-oriented search toolkit suitable for comparing optimal, heuristic, parallel, and machine-learning-flavored techniques on the same physical simulator.

Our contributions are:

- A faithful C++17 implementation of cluster tables, long-configuration dynamic programming, and tour-based fallback with parallel table precomputation.
- A `SearchStrategy` plugin architecture registering 34 solvers selectable via `--method`, including Knuth's Algorithm X (dancing links).

- Novel integrations: TDR zippers with configurable reset policies, SOM-guided best-first search, linear cube decomposition with SGD/backprop factorization, Lee–Seung NMF over cubee positional grams [8, 9], and DD parallel layer expansion.
- Developer-facing tooling: live ncurses cube visualization, progress reporting, reproducible random scrambles, and unit tests.

2 Background

2.1 Puzzle model

A puzzle instance is parameterized by `Dimensions` (c_1, c_2, n). The simulator tracks cubie positions and orientations on a $c_1 \times c_2 \times n$ grid with six colored faces. *Short moves* rotate slices perpendicular to the long z axis; *long moves* rotate x or y slices, permuting entire columns or rows of stacks.

2.2 Demaine Section 6 decomposition

Following [2], the key observations are:

1. Short moves affect only one *cluster* i, pairing slices $z = i$ and $z = n - 1 - i$.
2. Long moves induce a *long configuration*: an arrangement of $c_1 c_2$ stacks of length n. For fixed (c_1, c_2) the number of reachable long configurations is $O(1)$.
3. Short-move effects depend only on the current long configuration; per-cluster pattern tables encode repair costs.
4. An optimal solution uses only $O(1)$ long moves; the remainder are short moves within clusters.

2.3 Related search paradigms

Classical puzzle solvers rely on pattern databases and IDA* [7], two-phase algorithms [5], or group-theoretic decomposition. Our framework additionally exposes bidirectional zippered frontiers [4], beam search, Monte Carlo tree search with UCT [1], Kohonen map feature guidance [6], gradient-based latent factorization of cube features, and non-negative matrix factorization of positional-gram representations [8, 9] for empirical comparison rather than as production optimal solvers.

3 System and Algorithms

3.1 Architecture

The codebase separates *physical simulation* (`CubeState`, `Move`), *abstract state* (`LongConfig`, `ClusterSnapshot`), *optimal solver* (`OptimalSolver`), and *search plugins* (`SearchStrategy` via `SearchRegistry`). Table 1 summarizes registered CLI methods.

3.2 Optimal solver pipeline

Precomputation. `ShortMoveTable` builds a BFS over long configurations and, for each cluster i, a pattern table mapping (long index, cluster config) pairs to short-move distances. Cluster builds run in parallel across workers.

Search. Given start state s, the solver extracts long index and cluster snapshot, then runs DP/A* over long configurations with edge costs from cluster tables. If n exceeds a threshold,

Table 1: Representative search methods in `gods_solver`.

Category	Methods
Optimal (Demaine)	`demaine`, `dp`, `tour`, `exact-bfs`
Classical	`bfs`, `dfs`, `astar`, `ida`, `bdfs`, `algorithm-x`
Structured	`polytree-bfs/dfs`, `dp-bfs`, `zipper`
Parallel (DD)	`dd-bfs`, `dd-dfs`, `dd-zipper`, `dd-mcts`, `dd-som`, `dd-factor-sgd`, `dd-tdr-zipper`
Heuristic / ML	`fuzzy-pattern-db`, `mcts`, `som`, `factor-sgd`, `factor-bp`, `factor-nmf`
TDR Zipper	`tdr-zipper` (flags: variant, reset, bounded)

a *long-move tour* interleaves short repairs along a precomputed Hamiltonian path over long configs.

Complexity. For fixed (c_1, c_2): long-config BFS is $O(1)$; cluster table construction is $O(n)$ with parallelism; optimal search is $O(n \cdot \text{poly}(c_1, c_2))$.

3.3 Modular graph search

Each alternative method implements:

```
optional<Solution> solve(const CubeState& start,
                         const SearchEnvironment& env);
```

`SearchEnvironment` bundles dimensions, legal moves, goal state, caps (`max_depth`, `max_nodes`, `max_frontier`), heuristics, and an optional live `SearchProgress` dashboard.

Heuristics include zero, misplaced cubies, full pattern-database distance, and a *fuzzy* capped cluster sum for faster A*.

3.4 Tree-depth reset (TDR) zipper

Bidirectional `zipper` search [4] maintains forward and backward frontiers from scramble and goal, merging on hash collision. TDR extends this with periodic *tree-depth resets* controlled by `--zipper-reset`:

- **fixed**: reset every N layers;
- **random**: reset after a random interval in $[1, N]$;
- **entropy**: snapshot the frontier at peak width; restore on width collapse;
- **resample**: full restart from start/goal with a new RNG seed.

Variants (`--zipper-variant`) alter what is cleared: iterative-deepening style seen sets, frontier-only retention, polytree reordering, or abstract (long × cluster) keys. Optimal mode wraps phases in an outer iterative-deepening loop; bounded mode (`--zipper-bounded`) performs a single pass.

3.5 SOM-guided search

`som` extracts a normalized feature vector (face mismatch ratios, cubie error, cluster flags), maintains a 2D Kohonen map, and runs best-first search prioritizing moves that reduce feature distance to the solved state. `dd-som` runs independent maps in parallel and keeps the shortest solution.

3.6 Cube factorization (SGD / backprop)

`factor-sgd` and `factor-bp` decompose the same feature vector into a low-dimensional latent code z with a linear decoder $x \approx Wz+b$. The latent vector is partitioned into a *global* component and per-*cluster* slices, mirroring the Demaine cluster decomposition at the feature level. Given goal features $x^\star$, the model encodes $z^\star$ and minimizes

$$\mathcal{L}(x) = \|Wz + b - x\|_2^2 + \lambda\|z - z^\star\|_2^2$$

where z is obtained by a few gradient steps on the reconstruction term for fixed (W, b). `factor-sgd` updates (W, b, z) with stochastic gradient descent; `factor-bp` applies the same loss with explicit analytic backpropagation through encode and decode. Best-first search ranks legal moves by the predicted post-move loss. `dd-factor-sgd` runs independent factor models in parallel and retains the shortest solution. CLI flags `--factor-latent-dim`, `--factor-lr`, `--factor-steps`, and `--factor-encode-steps` control model capacity and training budget. Like SOM, these methods are exploratory: they solve small instances reliably but do not guarantee optimality under default node caps.

3.7 Positional grams and Lee–Seung NMF

`factor-nmf` maps cubee *positional grams*—non-negative unigram and adjacent-slot bigram indicators over fixed slot indices—into a vocabulary of roughly 3,000 dimensions on $2 \times 2 \times 3$ (12 slots $\times$ 12 cubies, plus adjacent bigrams). Goal-relative features are $\Delta = |v - v^\star|$, preserving non-negativity as required by NMF [8]. We factorize $\Delta \approx Wh$ with rank-k latent code $h \geq 0$ and basis $W \geq 0$, updating (W, h) via the Lee–Seung multiplicative rules [9] on the solved state and scramble before search. Best-first expansion ranks legal moves by positional-gram distance to the goal plus weighted latent alignment to the goal code $h^\star$. The `--factor-feature positional-gram` flag selects this backend for linear factorizers; `factor-nmf` enables it by default. On $2 \times 2 \times 3$ it previously achieved 5/5 success with 3–5 move solutions under a 500k node cap (Table 3 in earlier runs); at $3 \times 3 \times 3$ it timed out on all five scrambles within 180 s (Table 2).

3.8 Live solve UI

When stderr is a TTY, `SearchProgress` opens an ncurses session rendering a flattened cube net (U / L-F-R-B / D), expansion bar, elapsed time, and phase labels. An optional MIDI countdown precedes timed solves.

4 Benchmarks

We benchmark all 34 registered CLI methods on cubic puzzles $3 \times 3 \times 3$ (`c1=3, c2=3, n=3`) and $4 \times 4 \times 4$ (`c1=4, c2=4, n=4`). Each size uses five random scrambles (seeds 1–5; default scramble length $\max(4, \min(2n, 20))$), `--max-nodes 500000`, `--threads 1`, and progress disabled. Per-run wall-clock caps are 180 s (3^3) and 300 s (4^3). Table 2 lists success counts and mean solve time over successful runs (ms, or seconds when ≥ 10 s). "opt." / "subopt." mark methods that target optimal vs. exploratory solutions.

For reference, a prior run on the smaller $2 \times 2 \times 3$ toy instance (five 5-move scrambles) is summarized in Table 3.

Findings.

- On $3 \times 3 \times 3$, only `bdfs`, `zipper`, `dd-zipper`, `tdr-zipper`, and `dd-tdr-zipper` solved all five scrambles with sub-second mean latency; `astar` succeeded on all five but averaged 31 s.

Table 2: All methods on $3 \times 3 \times 3$ and $4 \times 4 \times 4$ (five scrambles each).

Method	3^3	Mean	4^3	Mean	
`algorithm-x`	0/5	–	0/5	–	opt.
`astar`	5/5	31.4 s	0/5	–	opt.
`bdfs`	5/5	25.0	5/5	1.3 s	opt.
`bfs`	1/5	64.9 s	0/5	–	opt.
`dd-bfs`	1/5	70.0 s	0/5	–	opt.
`dd-dfs`	1/5	64.9 s	0/5	–	opt.
`dd-factor-sgd`	0/5	–	0/5	–	subopt.
`dd-mcts`	0/5	–	0/5	–	subopt.
`dd-som`	0/5	–	0/5	–	subopt.
`dd-tdr-zipper`	5/5	65.3	5/5	15.4 s	opt.
`dd-zipper`	5/5	45.9	5/5	10.4 s	opt.
`demaine`	2/5	127.4 s	0/5	–	opt.
`dfs`	0/5	–	0/5	–	opt.
`dp`	1/5	75.6 s	0/5	–	opt.
`dp-bfs`	2/5	126.0 s	0/5	–	opt.
`exact-bfs`	1/5	74.6 s	0/5	–	opt.
`factor-bp`	0/5	–	0/5	–	subopt.
`factor-nmf`	0/5	–	0/5	–	subopt.
`factor-sgd`	0/5	–	0/5	–	subopt.
`fuzzy-pattern-db`	1/5	88.1 s	0/5	–	opt.
`ida`	0/5	–	0/5	–	opt.
`irregular-astar`	4/5	59.3 s	0/5	–	opt.
`irregular-bfs`	1/5	69.2 s	0/5	–	opt.
`irregular-ida`	0/5	–	0/5	–	opt.
`max-breadth-fs`	1/5	38.3 s	0/5	–	opt.
`mcts`	0/5	–	0/5	–	subopt.
`polytree-bfs`	1/5	91.1 s	0/5	–	opt.
`polytree-dfs`	0/5	–	0/5	–	opt.
`randomized-bfs`	0/5	–	0/5	–	opt.
`randomized-dfs`	0/5	–	0/5	–	opt.
`som`	0/5	–	0/5	–	subopt.
`tdr-zipper`	5/5	85.3	5/5	15.0 s	opt.
`tour`	0/5	–	0/5	–	opt.
`zipper`	5/5	58.3	5/5	10.2 s	opt.

- On $4 \times 4 \times 4$, the same five zipper/bidirectional family remained at 5/5; `bdfs` averaged 1.3 s while parallel DD zippers averaged 10–15 s. All other methods hit the 300 s cap or exhausted the node budget on every seed.
- Table-heavy Demaine paths (`demaine`, `dp`, `exact-bfs`) and unidirectional BFS variants succeed sporadically at 3^3 but not at 4^3 under these caps, illustrating the gap between polynomial-time existence and practical bounded search.
- Learning-guided methods (`mcts`, `som`, factorization, `algorithm-x`) did not finish within limits at either size; they remain exploratory baselines.
- The $2 \times 2 \times 3$ reference (Table 3) shows that many methods succeed on tiny instances; scaling to 3^3 and 4^3 separates structure-aware bidirectional search from exhaustive and table-build approaches.

5 Future Work

- **Larger n and exact benchmarking**: extend scaling studies to $5 \times 5 \times 5$ and beyond with cached tables and memory-mapped pattern databases; current 4^3 results show only bidirectional/zipper methods remain viable under fixed caps.
- **SAT / knowledge compilation**: encode reachability as CNF and compare DPLL, MCMC, and DNNF-based counting [10] against DP.

Table 3: Reference benchmark on $2 \times 2 \times 3$, five 5-move scrambles.

Method	Success	Mean (ms)	Min (ms)	Max (ms)	Notes
`astar`	5/5	9.7	5.2	15.0	optimal
`zipper`	5/5	8.6	6.9	9.5	optimal
`tdr-zipper`	5/5	10.6	7.7	12.4	optimal (ID)
`ida`	5/5	34.5	5.4	72.2	optimal
`mcts`	4/5	40.1	7.6	121.1	suboptimal (e.g. 9 moves)
`bfs`	5/5	112.6	9.8	202.2	optimal
`demaine`	5/5	129.3	10.4	230.7	optimal
`dp`	5/5	129.3	11.3	231.6	optimal
`exact-bfs`	5/5	130.7	10.8	236.7	optimal
`som`	5/5	950.9	416.2	1826.4	suboptimal (e.g. 21 moves)
`factor-nmf`	5/5	57241.3	2135.7	105944.2	positional-gram NMF; 3–5 moves

- **Learned heuristics**: train graph neural networks or reinforce MCTS priors on cluster-long features; extend linear factorization to nonlinear autoencoders with the same search wrapper.
- **Disk-resident frontiers**: spill BFS layers for TDR zippers on large state spaces.
- **Optimality certificates**: export witness paths from Demaine DP for formal verification.
- **Cross-method portfolio**: auto-select method by (c_1, c_2, n) and scramble depth using lightweight instance features.

6 Conclusion

We built a unified laboratory for generalized Rubik's cube algorithms spanning provably optimal Section 6 search and more than thirty alternative techniques. The implementation validates Demaine et al.'s polynomial framework on small instances, provides exact BFS baselines, and enables reproducible comparison of parallel, heuristic, SOM, MCTS, linear factorization, positional-gram NMF, Algorithm X (dancing links), and TDR zipper strategies through one CLI. Benchmarks on $3 \times 3 \times 3$ and $4 \times 4 \times 4$ (Table 2) show that bidirectional and zipper searches scale most reliably under fixed node and time caps, while table-heavy optimal methods and learning-guided search require larger budgets or smaller instances. The modular `SearchStrategy` design lowers the barrier to adding new solvers without touching the simulator core.

Acknowledgments

We thank the authors of [2] for the foundational algorithm and the open-source community for CMake, ncurses, and testing infrastructure.

References

[1] Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *Computers and Games*, pages 72–83. Springer, 2006.

[2] Erik D. Demaine, Martin L. Demaine, Sarah Eisenstat, Anna Lubiw, and Andrew Winslow. Algorithms for solving rubik's cubes. *arXiv preprint arXiv:1106.5736*, 2011. URL https://arxiv.org/abs/1106.5736.

[3] Erik D. Demaine, Martin L. Demaine, Sarah Eisenstat, Anna Lubiw, and Andrew Winslow. Algorithms for solving rubik's cubes. In *Proceedings of the 27th Annual Symposium on Computational Geometry (SoCG 2011)*, pages 689–704, 2011.

[4] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997. doi: 10.1017/S0956796897002864. URL https://en.wikipedia.org/wiki/Zipper_(data_structure)#CITEREFHuet1997.

[5] Herbert Kociemba. The two-phase algorithm and its extension to general twisting puzzles. *Symmetry: Culture and Science*, 3(1):471–484, 1992.

[6] Teuvo Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69, 1982.

[7] Richard E. Korf. Finding optimal solutions to rubik's cube using pattern databases. *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI)*, pages 700–705, 1997.

[8] Daniel D. Lee and H. Sebastian Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–791, 1999. doi: 10.1038/44565.

[9] Daniel D. Lee and H. Sebastian Seung. Algorithms for non-negative matrix factorization. In *Advances in Neural Information Processing Systems*, volume 13, pages 556–562, 2001.

[10] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 4 edition, 2020.