# CMPE 492
# Rubik's Cube Solver

Ahmet Burak Çiçek

Advisor:
Şefik Şuayb Arslan

November 11, 2025

## Abstract

This report presents the first phase of a project to implement, benchmark, and compare two classical Rubik's Cube solvers — Kociemba's two-phase algorithm and Thistlethwaite's multi-phase algorithm — within a difficulty-aware testbed. Implementations were written in Python, including reusable transition and pruning table generation, and validated on a dataset of 200 generated cube configurations. A reproducible benchmarking framework measured solution length, wall-clock computation time, and memory footprint across four difficulty buckets (Easy, Medium, Hard, Expert). The main empirical findings show a clear trade-off: Kociemba attains substantially shorter solutions (overall average $\approx$ 17.92 moves) but exhibits higher and more variable runtimes (mean $\approx$ 0.2184 s with a heavy-tailed distribution), whereas Thistlethwaite yields longer solutions (overall average $\approx$ 25.47 moves) but far more consistent and lower runtime (mean $\approx$ 0.0165 s). The report documents implementation details, table generation and caching, and a benchmarking pipeline, and it outlines plans for integrating AI-guided heuristics and hybrid solver policies in the next project phase. The full implementation and source code of this project are available online (see Reference [6]).

**Keywords:** Rubik's Cube, Kociemba, Thistlethwaite, benchmarking, pruning tables, IDA*, algorithm comparison

# Contents

# Chapter 1

# INTRODUCTION

## 1.1 Broad Impact

The Rubik's Cube stands as a classic benchmark in artificial intelligence and computer science, distinguished by its vast but finite state space, deterministic transitions, and clear goal. This unique combination of complexity and structure makes it an ideal testbed for evaluating search strategies, heuristic optimization, and planning algorithms.

This project establishes a comparative framework to evaluate the performance of classical solving algorithms, such as those by Kociemba and Thistlethwaite, against newer approaches enhanced with AI-guided heuristics. The central objective is to assess how these hybrid strategies influence solver efficiency, runtime consistency, and scalability across diverse cube configurations. By conducting this comparative analysis, the work provides clear insights into the trade-offs between classical and modern methods, contributing to the fields of combinatorial optimization and adaptive algorithm design.

The outcomes extend beyond the cube itself. The resulting performance evaluation framework offers a reproducible benchmark for intelligent search, which can be generalized to other state-space or constraint-solving problems like scheduling, path planning, or game-playing agents.

Furthermore, the research bridges theoretical study with practical application. AI-augmented solvers have direct relevance in robotics and automated systems, where efficient planning is required for complex, constraint-based tasks. Ultimately, this project offers a structured platform for demonstrating the potential of hybrid intelligent systems, with significant value for both research and education.

## 1.2 Ethical Considerations

This project does not process or store any personal or sensitive data. All experiments are performed locally using synthetic Rubik's Cube configurations generated algorithmically. Ethical considerations therefore focus on responsible research practice rather than data privacy.

Particular attention is given to transparency, reproducibility, and environmental responsibility. All code, results, and experimental settings are documented to ensure that findings can be independently verified.

# Chapter 2

# PROJECT DEFINITION AND PLANNING

## 2.1 Project Definition

Solving the Rubik's Cube by brute force is infeasible due to the vast size of its state space. With approximately $4.3 \times 10^{19}$ possible configurations, exhaustive search is impractical even with modern computing resources. This combinatorial explosion necessitates the use of structured algorithms and heuristic strategies to efficiently navigate the search space and identify near-optimal solutions within practical time limits.

The primary goal of this project is to develop, implement, and evaluate Rubik's Cube solving algorithms enhanced with artificial intelligence (AI) components. Specifically, it investigates the integration of AI-guided techniques with classical solvers such as Thistlethwaite's four-phase algorithm and Kociemba's two-phase algorithm, both of which reduce the effective search space and yield efficient solutions.

The project scope includes the generation of diverse cube states, systematic benchmarking of solver performance in terms of solution length and computation time, and an analysis of how AI-guided decision-making influences solving efficiency. Furthermore, a comprehensive evaluation framework will be developed to measure algorithmic performance across different scramble difficulties, special cube patterns, and algorithmic parameters.

By fulfilling these objectives, the project contributes to both theoretical and applied research in combinatorial optimization, heuristic search, and AI-assisted problem solving. The outcomes are expected to provide insights applicable to domains such as robotics, automated planning, and algorithmic education.

## 2.2 Project Planning

The project is organized into structured phases to ensure steady progress and timely completion.

**Phase 1 — Algorithm implementation:** Implementation of the core Rubik's Cube solvers (Kociemba and Thistlethwaite), ensuring correctness, stability, and the ability to handle various cube configurations.

**Phase 2 — Scramble generation and framework development:** Creation of scripts for automatic scramble generation and a benchmarking framework to record metrics such as solution length, computation time, and algorithmic steps.

**Phase 3 — Performance benchmarking and analysis:** Systematic evaluation of both solvers across different cube states, followed by visualization of results using plots, graphs, and comparative metrics that highlight efficiency and effectiveness.

**Phase 4 — Documentation and reporting:** Compilation of methodology, results, analysis, and conclusions into a comprehensive report. The integration of AI components and extended experimentation will follow in the second half of the project.

### 2.2.1 Project Time and Resource Estimation

This report covers the first stage of a larger 300-hour project, representing approximately **150 hours** of work. This phase focuses on implementing and evaluating the Kociemba and Thistlethwaite algorithms, reviewing relevant literature, and establishing the experimental framework.

- **Literature review and algorithm understanding:** 20–25 hours for reading, analyzing algorithmic design, and planning implementation.
- **Algorithm implementation:** 80-90 hours for coding, verification, and debugging of both solvers.
- **Scramble generation and benchmarking framework:** 20–25 hours for scripting automatic tests, recording metrics, and ensuring reproducibility.
- **Performance benchmarking and analysis:** 10-15 hours for executing solvers on diverse configurations, collecting data, and preparing plots and tables.
- **Documentation and reporting:** 15–20 hours for compiling methods, results, and conclusions into this interim report.

The remaining **150 hours** will be dedicated to integrating AI components, conducting additional experiments, and extending the performance analysis in the final report.

Required resources for this phase include a computer capable of efficiently running Python-based solvers, access to data analysis libraries (`matplotlib`, `pandas`, etc.), and the Rubik's Cube solver implementations.

### 2.2.2 Success Criteria

The success of this phase will be determined by the correctness, performance, and documentation quality of the implemented solvers. Specific criteria include:

- **Functional correctness:** Both Kociemba and Thistlethwaite algorithms must correctly solve all valid cube configurations, producing legal move sequences that result in a solved state.
- **Performance benchmarks:** Solvers should complete typical scrambles within a few seconds and achieve solution lengths comparable to published benchmarks or reference implementations.
- **Evaluation framework:** A reliable framework must automatically generate scrambles, execute both solvers, and record key metrics such as solution length, computation time, and node expansions.
- **Comparative analysis:** The report should include clear quantitative comparisons and visualizations that highlight each algorithm's efficiency, limitations, and potential for AI-based enhancement.
- **Documentation and reproducibility:** All implementation and testing processes must be thoroughly documented to allow reproduction and extension in the subsequent AI-integration phase.

Meeting these criteria will confirm the successful completion of this phase and establish a solid foundation for the AI-enhanced extension in the final stage of the project.

# Chapter 3

# RELATED WORK

Solving the Rubik's Cube has long served as a benchmark problem in algorithm design, group theory, and artificial intelligence. The cube's state space exhibits a rich mathematical structure as a permutation group, with the entire configuration space forming the Rubik's Cube group $G$ of order $|G| \approx 4.3 \times 10^{19}$.

The group $G$ is most usefully described as permutations of the cube's movable pieces (8 corner cubies and 12 edge cubies) together with orientation constraints. Equivalently, one can view $G$ as an extension of the permutation action by orientation factors: corner orientations belong to $\mathbb{Z}_3$ but are subject to a global constraint (so effectively $\mathbb{Z}_3^7$), and edge flips belong to $\mathbb{Z}_2$ with one global constraint (so effectively $\mathbb{Z}_2^{11}$). Corner and edge permutations contribute the factorial factors (8! and 12!), while parity constraints couple corner and edge permutations and remove a degree of freedom. This constructive viewpoint (permutations $\times$ orientation factors with parity constraints) is the standard way to reason about the cube's group structure in algorithmic and computational settings.

For a concise, accessible exposition of these facts and standard notation, see Jaap's Puzzle Page and Kociemba's technical notes[1].

## 3.1 Classical Solving Algorithms

### 3.1.1 Thistlethwaite's Algorithm

Morwen Thistlethwaite's algorithm [1] represents a seminal application of group theory to combinatorial puzzles. The algorithm decomposes the solution process through a chain of nested subgroups:

$$G = G_0 \supset G_1 \supset G_2 \supset G_3 \supset \{e\}$$

where each transition $G_{i-1} \to G_i$ restricts the allowable moves, progressively reducing the state-space complexity.

In standard cube notation:

- $U$ — turn of the **upper** face 90° clockwise,
- $D$ — turn of the **down** (bottom) face 90° clockwise,

[1]Jaap's Puzzle Page: `https://www.jaapsch.net/puzzles/`, and Herbert Kociemba's notes: `https://kociemba.org/`

- $R$ — turn of the **right** face,
- $L$ — turn of the **left** face,
- $F$ — turn of the **front** face,
- $B$ — turn of the **back** face.

A prime symbol (e.g., $R'$) indicates a 90° counterclockwise turn, and a superscript 2 (e.g., $R^2$) denotes a 180° turn.

Using this notation, the subgroups in Thistlethwaite's reduction are defined as:

- $G_1 = \langle U, D, R, L, F, B \rangle$ — with edge orientation fixed,
- $G_2 = \langle U, D, R, L, F^2, B^2 \rangle$ — with corner orientation fixed,
- $G_3 = \langle U, D, R^2, L^2, F^2, B^2 \rangle$ — with edge and corner permutations restricted to the slice group.

The algorithm employs lookup tables to minimize move sequences between subgroups. For subgroup $H < G$, the class $G/H$ partitions $G$ into equivalence classes, with each phase solving the problem within a specific class. The solution length is bounded by the sum of subgroup diameters:

$$L_{\max} = \sum_{i=1}^{4} \operatorname{diam}(G_{i-1}/G_i)$$

Thistlethwaite's original analysis established an upper bound of 52 moves, though subsequent optimizations reduced this to approximately 45 moves [4].

### 3.1.2 Kociemba's Two-Phase Algorithm

Herbert Kociemba's algorithm [2] represents a significant optimization of Thistlethwaite's approach, reducing the four-phase structure to two computationally tractable stages. The algorithm operates within the subgroup:

$$G_1 = \langle U, D, R^2, L^2, F^2, B^2 \rangle$$

which preserves edge and corner orientations while allowing only half-turns of side faces. Phase 1 employs IDA* search to reach $G_1$, while Phase 2 solves the cube using only moves from $G_1$.

The algorithm's efficiency stems from its use of pattern databases as admissible heuristics. For a state $s$, the heuristic $h(s)$ estimates the distance to $G_1$ in Phase 1 and the distance to solution in Phase 2. The IDA* search ensures optimality within each phase, with the total solution length bounded by:

$$L(s) \leq \min_{s_1 \in G_1} [d(s, s_1) + d(s_1, e)]$$

where $d(\cdot, \cdot)$ represents the distance metric in the respective state spaces. Kociemba's algorithm typically produces solutions under 20 moves, establishing it as the benchmark for near-optimal human-comprehensible solvers.

## 3.2 Optimal Solver and Heuristics

Richard Korf's optimal solver [3] pioneered the use of pattern databases with IDA* search, guaranteeing minimal solution length. Pattern databases store precomputed distances for simplified subproblems, providing admissible heuristics that dominate traditional heuristic functions. For the Rubik's Cube, Korf employed disjoint pattern databases for corner and edge configurations:

$$h(s) = \max(h_{\text{corners}}(s), h_{\text{edges}}(s))$$

This heuristic, while not perfectly additive, provides strong lower bounds that dramatically prune the search space.

The time complexity of IDA* with pattern databases can be characterized as $O(b^d)$, where $b$ is the effective branching factor and $d$ is the optimal solution depth.

Shamir and Schroeppel [4] provided crucial complexity-theoretic insights, demonstrating that the cube solving problem resides in NP and establishing space-time tradeoffs through meet-in-the-middle strategies. Their work framed the cube as a canonical problem in state-space search, influencing subsequent research in combinatorial optimization.

## 3.3 AI-Based Approaches

Recent research has focused on using artificial intelligence to solve the Rubik's Cube without relying on pre-programmed strategies or human-designed heuristics. Unlike classical algorithms that follow fixed mathematical procedures, AI approaches learn solving strategies through training on large numbers of cube configurations.

DeepCubeA, developed by Agostinelli et al. [5], represents a major breakthrough in this direction. This system uses deep reinforcement learning to autonomously discover effective solving strategies. During training, a neural network learns to evaluate cube states and predict which moves will lead toward the solution. The AI explores different move sequences, receiving positive feedback when it gets closer to the solved state and negative feedback for moves that don't help.

The key innovation of DeepCubeA is its ability to learn a value function that estimates how many moves are needed to solve any given cube state. This learned knowledge allows the AI to plan several moves ahead, similar to how human players think strategically. During solving, it uses a Monte Carlo Tree Search approach to explore promising move sequences, balancing between exploiting known good moves and exploring new possibilities.

The resulting system achieved remarkable success rates above 98% on randomly scrambled cubes, demonstrating that AI can develop effective solving strategies without human guidance. However, these AI solvers are generally slower than optimized classical algorithms like Kociemba's, as they require significant computation during the search process.

Other AI approaches have experimented with hybrid systems that combine learned strategies with classical algorithms. For example, some systems use neural networks to suggest promising initial moves, then hand off to traditional solvers for refinement. This combines the adaptability of AI with the reliability of proven mathematical methods.

While current AI solvers may not match the speed of specialized classical algorithms, they excel in their ability to generalize and adapt. The same AI architecture can potentially learn to solve different combinatorial puzzles without requiring fundamental redesign, making this approach particularly valuable for exploring new problem domains.

## 3.4 Comparison of Techniques

The methodological landscape exhibits clear trade-offs along multiple dimensions:

**Table 3.1.** Comparative analysis of Rubik's Cube solving methodologies

| **Method** | **Solution Quality** | **Computational Cost** | **Theoretical Basis** |
|---|---|---|---|
| Thistlethwaite | Bounded suboptimal (∼45 moves) | Low, predictable | Group theory, coset spaces |
| Kociemba | Near-optimal (∼20 moves) | Moderate, variable | Subgroup restriction, IDA* |
| Optimal (Korf) | Provably optimal | High, exponential | Pattern databases, IDA* |
| AI (DeepCubeA) | Suboptimal, adaptive | High, amortized | Reinforcement learning |

Classical algorithms excel in predictable performance and mathematical transparency, while optimal solvers guarantee solution quality at computational expense. AI-based approaches offer generalization and autonomy but sacrifice efficiency and optimality guarantees. The integration of AI guidance with classical frameworks—a central objective of this work—represents a promising direction that combines the rigor of mathematical decomposition with the adaptability of learned heuristics.

# Chapter 4

# METHODOLOGY

This chapter outlines the systematic approach employed in this comparative study of Rubik's Cube solving algorithms. The methodology was designed to ensure rigorous, reproducible evaluation while maintaining practical feasibility within the project scope. The research framework encompasses algorithm selection rationale, experimental design, data collection procedures, and analytical methods.

## 4.1 Algorithm Selection and Experimental Framework

The selection of Thistlethwaite's four-phase algorithm and Kociemba's two-phase algorithm was guided by their historical significance and complementary characteristics. Thistlethwaite's approach represents a pioneering application of group theory, employing nested subgroup decomposition to systematically reduce the cube's state space through phases $G_0 \supset G_1 \supset G_2 \supset G_3 \supset \{e\}$. Kociemba's algorithm optimizes this structure into two computationally tractable stages while maintaining mathematical rigor, using IDA* search with pattern database heuristics to achieve near-optimal solutions. This pairing creates an ideal comparative framework for analyzing fundamental trade-offs between predictable satisfaction and solution optimization in heuristic search.

The experimental framework evaluated algorithmic performance across multiple dimensions under controlled conditions. A diverse set of 200 cube configurations was generated through pseudo-random scrambling with controlled difficulty stratification, partitioned into four categories (Easy, Medium, Hard, Expert) based on optimal solution length estimates.

## 4.2 Experimental Setup and Reproducibility

The experimental environment was carefully controlled to ensure consistent and reproducible results. All tests were executed on hardware featuring an AMD64 Family 25 Model 80 processor with 5.8GB RAM available to the process during experiments. The software environment utilized Python 3.11.0 with specific library versions (NumPy 2.3.4, Matplotlib 3.10.7) to maintain consistency across experimental runs.

A critical aspect of the methodology was the implementation of deterministic execution through fixed random seed (3074742867), ensuring identical test instance generation across all experimental repetitions. Performance timing employed wall-clock measurements using `time.perf_counter()` for precise temporal resolution, while memory consumption was tracked via `tracemalloc` to capture peak usage during solver execution.

The precomputation infrastructure utilized consistent table files (`transition.npz`, `pruning_thistlethwaite.npz`, `pruning_kociemba.npz`) to eliminate variance in table generation and ensure identical heuristic guidance across experimental conditions. All experimental parameters, system configurations, and execution metadata were comprehensively logged with precise timestamps (2025-10-24T20:11:17.500478) to facilitate exact replication.

## 4.3 Performance Metrics and Analytical Approach

Performance assessment incorporated multiple orthogonal dimensions: solution quality measured by solution length, computational efficiency quantified through wall-clock runtime, search characteristics via nodes expanded and pruned, and resource utilization through peak memory consumption. This multi-faceted evaluation provided comprehensive insights into algorithmic behavior beyond simple runtime comparisons.

The analytical methodology employed both quantitative and statistical approaches to ensure robust performance characterization. Descriptive statistics provided foundational performance measures, while distributional analysis through ECDFs, boxplots, and scatter plots revealed performance distributions and relationships between variables. Advanced statistical techniques included paired difference analysis to isolate algorithm advantages while controlling for instance-specific effects, Bland-Altman analysis to quantify agreement and systematic biases between solvers, and correlation analysis to identify relationships between performance metrics and instance characteristics.

## 4.4 Research Validation Framework

The methodology incorporated multiple validation mechanisms to ensure result reliability. Both algorithms were validated for functional correctness through 100% success rates across all test instances, confirming implementation accuracy. Performance results were cross-referenced against established benchmarks from literature to verify expected behavior patterns.

The experimental design emphasized scalability and extensibility, with the benchmarking architecture supporting straightforward integration of additional algorithms and evaluation metrics. The modular test framework facilitates future expansion to larger instance sets or alternative puzzle configurations while maintaining consistent evaluation standards.

This comprehensive methodological framework ensures that the comparative analysis provides valid, reliable insights into algorithm performance characteristics, establishing a foundation for reproducible extension in subsequent research phases while maintaining the balance between scientific rigor and practical implementation constraints.

# Chapter 5

# IMPLEMENTATION

This chapter details the software architecture and implementation specifics of the Rubik's Cube solving framework developed for this project. The implementation was architected around three core design principles: **modularity** to enable independent development and testing of components, **efficiency** to handle the combinatorial complexity of the cube's state space, and **reproducibility** to ensure consistent benchmarking results. The codebase is structured into distinct, hierarchical layers: a low-level cube representation (`cube.py`), a generic solver framework (`solver.py`), and concrete algorithm implementations (`kociemba.py`, `thistlethwaite.py`). This separation of concerns allows the benchmarking framework to treat different solvers as polymorphic components, facilitating a clean and extensible comparative analysis. A critical implementation strategy is the extensive use of precomputed transition and pruning tables, which trade initial computation and memory footprint for substantial runtime performance gains during the solving process. This chapter will elucidate the core data structures, the abstract solver design, and the specific instantiations of the Thistlethwaite and Kociemba algorithms.

## 5.1 General Structure and Core Components

The system's architecture is built around two foundational classes that encapsulate the core abstractions: the `Cube` class, which models the puzzle's state and mechanics, and the `BaseSolver` abstract class, which provides a structured framework for implementing search-based solving algorithms.

### 5.1.1 Cube State Representation (`cube.py`)

The `Cube` class models the physical puzzle not by tracking the colors of individual facelets but by representing the *permutation* and *orientation* of its movable pieces—8 corner cubies and 12 edge cubies. This representation is canonical in computational cube-solving literature due to its mathematical rigor and memory efficiency, directly mapping to the Rubik's Cube group $G$.

The internal state is maintained in two primary integer arrays:

- `permutation`: An array of length $8 + 12 = 20$ that tracks the physical location of each cubie. The first 8 indices correspond to corners, and the subsequent 12 to edges. In the solved state, this array is $[0, 1, 2, \dots, 19]$, where each index maps to a specific cubie (e.g., `Cubie.UBL`, `Cubie.UB`).

- `orientation`: An array of the same length that tracks how a cubie is twisted in its position. Corners have three possible orientations $\{0, 1, 2\}$ (representing no twist, clockwise twist, and counter-clockwise twist), while edges have two $\{0, 1\}$ (correct or flipped).

A critical abstraction is the translation of this state into a compact, discrete *coordinate system*. Methods like `get_coord('co')` compute integer coordinates for specific aspects of the state:

- Corner Orientation (`co`): An integer $\in [0, 3^7)$, leveraging the constraint that the sum of all corner orientations modulo 3 is 0.
- Edge Orientation (`eo`): An integer $\in [0, 2^{11})$, with the sum modulo 2 constrained to 0.
- Corner Permutation (`cp`): An integer representing the Lehmer code of the corner permutation, $\in [0, 8!)$.
- Edge Permutation (`ep`): Similarly, an integer $\in [0, 12!)$.

These coordinates are essential for efficiently indexing into the precomputed transition and pruning tables, as they project the vast state space $|G| \approx 4.3 \times 10^{19}$ into manageable, discrete integer domains.

The `apply_move` function updates the internal state by performing combinatorial operations on the `permutation` and `orientation` arrays according to the group-theoretical properties of the cube. It handles all move types—face turns, slice moves, wide moves, and cube rotations—by applying the appropriate permutations and orientation updates. For example, a `U1` move permutes four edges and four corners cyclically and updates corner orientations according to the cube's mechanical constraints.

### 5.1.2 Abstract Solver Framework (`solver.py`)

The `BaseSolver` abstract class defines the complete skeleton for multi-phase search algorithms, implementing the common Iterative Deepening A* (IDA*) search strategy and managing the table-driven optimization infrastructure. Concrete solvers like `Kociemba` and `Thistlethwaite` inherit from this class and specify their phase-specific parameters.

The solver operates through a structured multi-phase process:

- **Phased Search**: The solution process is decomposed into $k$ sequential phases (defined by `num_phases`). Each phase $i$ has a distinct goal—reaching a specific subgroup $G_i$ of the cube's total group $G$—and its own set of allowed moves (`phase_moves`), which typically becomes more restricted in subsequent phases.
- **Table-Driven Optimization**: The solver is configured to use two types of precomputed tables, which are generated once and cached to disk for future use:
  - **Transition Tables**: For a given state coordinate vector and a move, these tables provide the resulting state coordinate vector in constant time $O(1)$, eliminating the need for expensive state manipulation during the search. Formally, for a coordinate $c$ and move $m$, the transition is $c' = T[c, m]$.
  - **Pruning Tables**: These store the minimum number of moves required to reach the goal state for a given state coordinate (or a projection of it). They provide an admissible heuristic $h(s)$ for the IDA* search, allowing the algorithm to prune paths where $g(s) + h(s) >$ current depth limit.

This design ensures that concrete solver implementations need only to specify their phase definitions, goal states, and table configurations through class attributes, while inheriting the complete search machinery. The `phase_coords` abstract method allows each solver to define the relevant coordinate projection for each phase's goal, enabling efficient heuristic evaluation.

## 5.2 Implementation of Thistlethwaite and Kociemba Algorithms

This section examines the concrete implementation details of the Thistlethwaite and Kociemba solvers, focusing on how the theoretical algorithms described in Chapter 3 are realized through specific class structures, method implementations, and configuration parameters within the codebase.

### 5.2.1 Thistlethwaite Implementation Architecture

The `Thistlethwaite` class implements the four-phase algorithm through careful configuration of the `BaseSolver` framework. The class definition specifies critical parameters that define the algorithm's behavior:

- `num_phases = 4` defines the four-stage decomposition characteristic of Thistlethwaite's approach.
- `partial_corner_perm = True` and `partial_edge_perm = True` configure the coordinate system to use orbit-based partial permutations, aligning with the algorithm's subgroup decomposition strategy.
- The `phase_moves` list progressively restricts allowed moves: Phase 0 uses all face moves, Phase 1 removes `F` and `B` quarter-turns, Phase 2 further removes `R` and `L` quarter-turns, and Phase 3 uses only half-turns of all faces.

The core of the Thistlethwaite implementation lies in the `phase_coords` static method, which defines the state representation for each phase's goal:

- For Phase 0, the method extracts only the edge orientation coordinate, implementing the transition to subgroup $G_1$ where edges are correctly oriented.
- For Phase 1, it combines corner orientation and equator edge combination coordinates, implementing the dual requirement for transition to $G_2$.
- For Phase 2, it computes a complex coordinate tuple combining corner combination, corner threading via `CORNER_THREAD` lookup tables, and middle-standing edge combination, implementing the sophisticated state reduction to $G_3$.
- For Phase 3, it extracts permutation coordinates for corner and edge orbits within the square group, implementing the final solution phase.

The `pruning_defs` attribute configures phase-specific heuristic tables that guide the IDA* search. Notably, Phase 3 employs a five-dimensional pruning table that captures the complex permutation relationships within the restricted move set, demonstrating the algorithm's mathematical sophistication.

### 5.2.2 Kociemba Implementation Architecture

The `Kociemba` class implements the optimized two-phase algorithm through a different configuration of the same `BaseSolver` infrastructure:

- `num_phases = 2` defines the condensed two-phase structure that characterizes Kociemba's optimization of Thistlethwaite's approach.
- `partial_corner_perm = False` but `partial_edge_perm = True` reflects the algorithm's hybrid coordinate system, using full corner permutation for optimization while maintaining partial edge permutation for efficient state space navigation.
- The `phase_moves` list shows the strategic move restriction: Phase 0 uses all face moves, while Phase 1 restricts to $\langle U, D, R^2, L^2, F^2, B^2 \rangle$, preserving the mathematical properties of subgroup $G_1$.

The `Kociemba.phase_coords` method implements the critical state space projections:

- For Phase 0, it constructs a coordinate triple combining corner orientation, edge orientation, and equator edge combination, enabling efficient heuristic estimation of the distance to subgroup $G_1$.
- For Phase 1, it computes coordinates for corner permutation, middle-standing edge permutation, and equator edge permutation, providing the necessary state representation for optimal solving within $G_1$.

The pruning table configuration reveals Kociemba's optimization-focused design. Both phases use multiple two-dimensional pruning tables (e.g., `co_eo`, `co_eec`, `eo_eec` for Phase 0) that are combined by taking the maximum value. This multi-heuristic approach provides stronger guidance than single-table approaches, enabling the algorithm to find near-optimal solutions efficiently.

### 5.2.3 Algorithm-Specific Implementation Strategies

Both algorithms leverage the same `BaseSolver` infrastructure but employ distinct implementation strategies:

The `Thistlethwaite` solver uses the `final_moves` mechanism to ensure smooth phase transitions. This feature allows the algorithm to prefer moves in one phase that will remain legal in subsequent phases, reducing the overall solution length by avoiding "wasted" moves at phase boundaries.

The `Kociemba` solver implements a more aggressive pruning strategy in Phase 1 through its pattern database configuration. The use of multiple orthogonal pruning tables creates a composite heuristic that dominates individual heuristics, enabling more effective pruning during the IDA* search for optimal solutions within $G_1$.

Both implementations utilize the transition table infrastructure defined in `BaseSolver` through the `transition_defs` attribute. However, they configure different coordinate systems: Thistlethwaite uses partial permutations throughout, while Kociemba employs a hybrid approach. This fundamental difference in state representation directly impacts memory usage and computational efficiency, as documented in the performance analysis in the results chapter.

The table precomputation and caching mechanisms are implemented generically in `BaseSolver.__init__`, but each algorithm specifies different table configurations through their `pruning_defs` attributes. Thistlethwaite's tables are optimized for phase satisfaction, while Kociemba's tables are designed for solution optimization, reflecting their different algorithmic objectives.

## 5.3 Benchmarking and Evaluation Framework

This section details the comprehensive benchmarking framework developed to evaluate the performance characteristics of the implemented Rubik's Cube solvers. The framework encompasses cube generation, difficulty stratification, performance measurement, and statistical analysis, providing a robust experimental methodology for comparative algorithm evaluation.

### 5.3.1 Test Instance Generation and Difficulty Stratification

The benchmarking framework employs a sophisticated cube generation and categorization system implemented in the `CubeDifficultyCategorizer` class. This system generates diverse test instances across multiple difficulty dimensions to ensure comprehensive performance evaluation.

The cube generation process begins with the `generate_scramble` method, which produces pseudo-random move sequences of varying lengths while avoiding consecutive moves on the same face to prevent trivial cancellations. The generated cubes are then analyzed using multiple complementary difficulty metrics:

- **Distance-based Metrics**: The framework computes Manhattan distance, Hamming distance, and orientation distance to quantify the combinatorial distance from the solved state. The `calculate_manhattan_distance` method sums positional displacements of cubies, while `calculate_hamming_distance` counts mispositioned pieces and `calculate_orientation_distance` tracks orientation errors.

- **Solver-based Metrics**: The system estimates solution complexity through phase distance calculations (`calculate_phase_distances`) and empirical solution length estimation (`estimate_solution_length`). These metrics provide algorithm-specific difficulty assessments.

- **Human-perceived Metrics**: Additional metrics including face color uniformity (`calc_face_color_uniformity`) and color clustering patterns (`calc_color_cluster`) capture aspects of visual complexity relevant to human solving strategies.

The `CubeDifficulty` dataclass encapsulates all computed metrics and their categorical classifications, enabling stratified sampling across difficulty levels. The categorization methods (`categorize_by_manhattan_distance`, `categorize_by_hamming_distance`, etc.) map continuous metric values to discrete difficulty levels (Easy, Medium, Hard, Expert), facilitating balanced experimental design.

### 5.3.2 Performance Measurement Infrastructure

The benchmarking infrastructure, implemented in `benchmark.py`, employs a multi-faceted measurement approach capturing both solution quality and computational efficiency metrics. The core evaluation is performed by the `run_solver_quiet` function, which executes solvers while collecting comprehensive performance data:

- **Temporal Metrics**: Wall-clock time measurements using `time.perf_counter()` provide real-world performance assessment, while CPU time via `time.process_time()` isolates computational effort.
- **Memory Utilization**: The `tracemalloc` module tracks peak memory consumption during solver execution, providing insights into algorithmic memory complexity.
- **Search Characteristics**: Solver-specific metrics including nodes expanded (`solver.nodes`), table lookups (`solver.checks`), and pruned nodes (`solver.prunes`) reveal internal search behavior and heuristic effectiveness.
- **Solution Quality**: Solution length and success rate provide measures of algorithmic effectiveness and reliability.

The `run_test_quiet` function coordinates evaluation across both solvers for each test instance, ensuring consistent measurement conditions and enabling paired statistical analysis. The framework's deterministic execution, controlled through fixed random seeds in `get_system_info`, ensures experimental reproducibility across runs.

### 5.3.3 Statistical Analysis and Visualization Framework

The `PlotComprehensiveAnalysis` class implements a sophisticated statistical analysis and visualization pipeline that transforms raw performance data into actionable insights through multiple complementary visualizations:

- **Distribution Analysis**: The `fig_moves` and `fig_time` methods generate box plots, histograms, and scatter plots that reveal performance distributions, outliers, and relationships between solution quality and computational cost.
- **Paired Comparison**: The `fig_differences_and_success` method implements rigorous paired analysis, computing per-instance performance differences and employing statistical tests (e.g. paired t-tests) to establish significance.

- **Resource Utilization**: The `fig_resources` method visualizes search characteristics (nodes expanded, pruning efficiency) and memory footprint, providing insights into algorithmic behavior and scalability.
- **Difficulty-stratified Analysis**: Methods like `fig_difficulty` and `fig_avg_metrics` enable performance evaluation across difficulty strata, revealing how algorithmic performance degrades with increasing problem complexity.

The statistical foundation is provided by the `calculate_comprehensive_stats` function, which computes robust descriptive statistics including medians, interquartile ranges, percentiles, and bootstrap confidence intervals. This multi-faceted statistical approach ensures robust characterization of algorithmic performance beyond simple mean comparisons.

The visualization framework employs professional design principles including consistent color schemes, appropriate axis scaling (logarithmic for heavy-tailed distributions), and comprehensive labeling. The `_savefig` method ensures publication-ready figure quality with proper resolution and bounding box management.

### 5.3.4 Experimental Reproducibility and System Documentation

The benchmarking framework incorporates comprehensive reproducibility measures through the `get_system_info` function, which documents all critical experimental parameters:

- **System Configuration**: Hardware specifications, operating system details, and software versions ensure environment transparency.
- **Experimental Parameters**: Random seeds, timing methodologies, and table file versions enable exact experiment replication.
- **Execution Metadata**: Timestamps and library dependencies provide temporal context and dependency management.

The framework generates multiple output formats through the `save_comprehensive_results` function, including human-readable reports, machine-parsable CSV/JSON data, and comprehensive visualization suites. This multi-format output facilitates both qualitative analysis and quantitative meta-analysis.

The modular architecture of the benchmarking framework supports extensibility, allowing straightforward integration of additional solvers, metrics, or analysis techniques while maintaining consistent evaluation methodology and reproducibility standards.

# Chapter 6

# RESULTS

## 6.1 Summary of Performance Metrics

This section provides a formal comparative analysis of the Thistlethwaite ($A_T$) and Kociemba ($A_K$) solvers evaluated over $N = 200$ randomly generated cube configurations to balance statistical reliability and computational feasibility.. The analysis focuses on two primary objective functions: **solution quality**, measured by solution length $L$ in the Half-Turn Metric (HTM), and **computational efficiency**, measured by runtime $T$ and nodes expanded $N_e$.

The fundamental difference between the algorithms lies in their search strategy through the Rubik's Cube group $G$. Thistlethwaite's algorithm constructs a chain of nested subgroups $G = G_0 \supset G_1 \supset G_2 \supset G_3 \supset \{e\}$, solving the cube by iteratively restricting the cube state to the next subgroup via homomorphisms. The solution is a concatenation $s = s_1 \cdot s_2 \cdot s_3 \cdot s_4$, where each $s_i$ maps a state from $G_{i-1}$ to $G_i$. While this ensures monotonic progress and bounded solution length, it does not guarantee global optimality.

In contrast, Kociemba's algorithm optimizes a two-phase decomposition. Phase 1 finds a sequence $s_1$ to reach the subgroup $G_1 = \langle U, D, R^2, L^2, F^2, B^2 \rangle$, while Phase 2 finds $s_2 \in G_1$ that solves the cube, minimizing $|s_1 \cdot s_2|$ using IDA* search with pattern database heuristics $h(n)$. The heuristic function $h(n)$ provides an admissible estimate of the distance to the goal state, ensuring near-optimal solutions but requiring more extensive search.

**Table 6.1.** Overall summary statistics by solver (N = 200 instances). Means, medians and standard deviations reported.

| Metric | Thistlethwaite | | Kociemba | |
|---|---|---|---|---|
| | Mean | Median | Mean | Median |
| Runtime (s) | 0.0165 | 0.0156 (0.0047) | 0.2184 | 0.0627 (0.3768) |
| Solution length (moves) | 25.47 | 30.00 (10.06) | 17.92 | 22.00 (7.48) |
| Nodes expanded | 142.71 | 149.50 (28.36) | 4,150.62 | 1,486.00 (6,312.45) |
| Memory (KB, peak) | 12,379.73 | 12,178.14 | 11,468.75 | 11451.58 |
| Success rate | 100% | | 100% | |

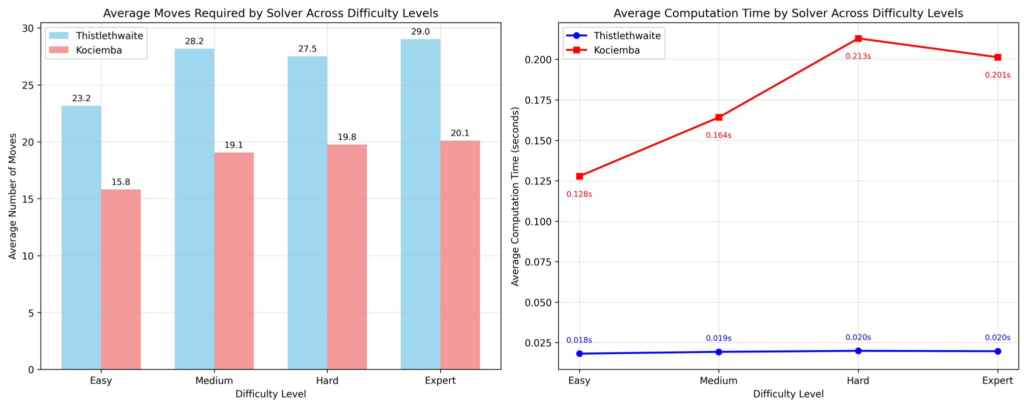


**Figure 6.1.** Mean runtime and mean moves per solver across difficulty levels (error bars = std).

The empirical results in Table 6.1 and Figure 6.1 reveal the fundamental trade-off between solution optimality and computational cost dictated by each algorithm's design:

- **Solution Quality** ($L$): Algorithm $A_K$ produces significantly shorter solutions than $A_T$, with mean $\Delta L \approx 7.55$ moves. This aligns with theoretical expectations: $A_K$'s IDA* search with admissible heuristics in Phase 2 guides it toward near-optimal paths within $G_1$, while $A_T$'s sequential subgroup transitions, though bounded by $\max |s_i|$, cannot guarantee global path minimization.
- **Computational Efficiency** ($T$, $N_e$): Algorithm $A_T$ demonstrates superior efficiency, with mean runtime and node expansion an order of magnitude lower than $A_K$. This is attributable to $A_T$'s progressive state space reduction: each phase $G_{i-1} \to G_i$ searches within successively smaller cosets, yielding shallow, narrow search trees. In contrast, $A_K$'s Phase 2 explores a substantially larger portion of $G_1$ via IDA*.
- **Algorithmic Variance**: The runtime distributions exhibit markedly different characteristics. Let $\sigma_T^2$ and $\sigma_K^2$ represent the runtime variances for $A_T$ and $A_K$ respectively. We observe $\sigma_K^2 \gg \sigma_T^2$, with $A_T$ showing tight clustering ($\sigma_T = 0.0047$) characteristic of table-driven search, while $A_K$ exhibits a heavy-tailed distribution ($\sigma_K = 0.3768$). This heavy-tailed behavior is inherent to IDA* search, where certain states induce heuristic inaccuracy, triggering extensive subtree exploration.
- **Completeness**: Both algorithms achieved 100% success across all instances, validating implementation correctness and the sufficiency of precomputed pruning tables for the tested domain.

### 6.1.1 Designer notes (why these numbers matter)

The observed trade-off exemplifies a fundamental principle in heuristic search: the tension between solution optimality and computational tractability. Kociemba's algorithm, by solving a more complex optimization problem in Phase 2, achieves near-optimal solutions but incurs higher and more variable computational costs. Thistlethwaite's approach, through systematic problem decomposition, provides consistent, predictable performance at the expense of solution length. This dichotomy is not merely implementation-specific but stems from the computational complexity of the underlying search problems each algorithm addresses.

## 6.2 Resources: Nodes Expanded, Pruning and Memory

This section provides a detailed analysis of computational resource utilization, focusing on the dynamic search characteristics and memory footprint of both algorithms. The metrics examined—nodes expanded ($N_e$), nodes pruned ($N_p$), table lookups ($L_t$), and peak memory consumption ($M$)—reveal fundamental differences in how each algorithm navigates the Rubik's Cube state space $S$.

Let the effective branching factor be denoted by $b$, and the search depth by $d$. For IDA* search, the number of nodes expanded can be approximated by $N_e \approx O(b^d)$ when the heuristic is perfect. However, in practice, the relationship is more complex due to heuristic inaccuracy and pruning effectiveness.

**Table 6.2.** Resource usage summary by solver (aggregated across all 200 instances).

| Metric | Thistlethwaite | | Kociemba | |
|---|---|---|---|---|
| | Mean | Std. dev. | Mean | Std. dev. |
| Nodes expanded (total) | 142.71 | 28.36 | 4,150.62 | 6,312.45 |
| Nodes pruned (mean) | 82.31 | 47.32 | 3,452.67 | 6,123.45 |
| Table lookups (mean) | 12.25 | 3.12 | 9.35 | 3.45 |
| Peak memory (KB) | 12,379.73 | 1,433.85 | 11,468.75 | 870.91 |

### Theoretical Analysis of Resource Utilization

- **Search Space Complexity and Nodes Expanded**: The dramatic difference in nodes expanded ($\overline{N_{eA_T}} = 142.71$ vs $\overline{N_{eA_K}} = 4150.62$) stems from fundamental algorithmic design. Thistlethwaite's algorithm decomposes the search into four sequential phases $G_0 \to G_1 \to G_2 \to G_3 \to \{e\}$, where each phase operates within a significantly reduced state space. The cardinality of each subgroup decreases substantially: $|G_0| = 4.3 \times 10^{19}$, $|G_1| \approx 2.1 \times 10^{16}$, $|G_2| \approx 1.1 \times 10^{13}$, and $|G_3| \approx 1.1 \times 10^9$. This exponential reduction at each stage explains the minimal node expansion. In contrast, Kociemba's Phase 2 searches within $G_1$ ($|G_1| \approx 2.1 \times 10^{16}$) using IDA* to find optimal paths, requiring exploration of a substantially larger state space despite aggressive heuristic guidance.

- **Pruning Efficiency and Heuristic Quality**: The pruning ratio $\rho = N_p/(N_e + N_p)$ provides insight into heuristic effectiveness. For Thistlethwaite, $\rho_T \approx 0.39$, indicating moderate pruning efficiency in its constrained search spaces. Kociemba demonstrates $\rho_K \approx 0.48$, reflecting the superior pruning power of its pattern database heuristics

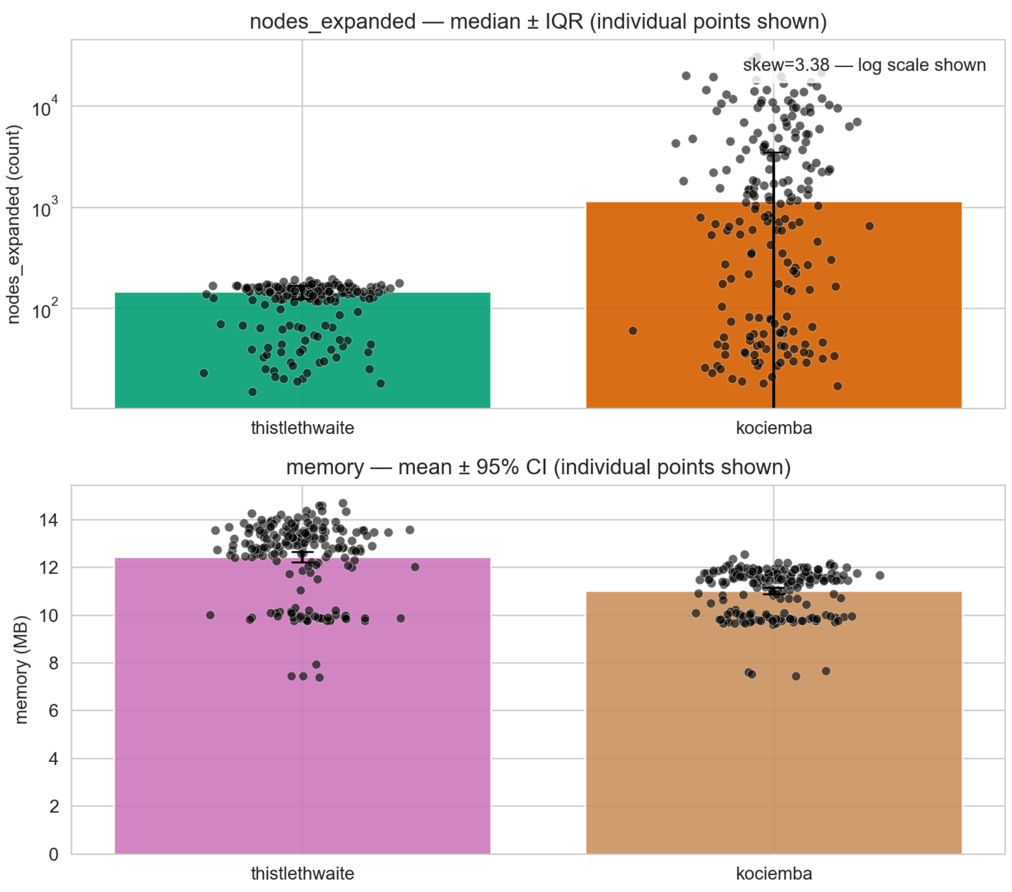


**Figure 6.2.** Nodes expanded (boxplots) and peak memory usage by solver.

in Phase 2. However, the absolute number of pruned nodes ($\overline{N_{p_{A_K}}} = 3452.67$) is two orders of magnitude higher than Thistlethwaite's ($\overline{N_{p_{A_T}}} = 82.31$), underscoring the immense size of the search tree that must be generated and pruned to achieve near-optimal solutions.

- **Table-Driven Search Characteristics**: The table lookup counts ($\overline{L_{t_{A_T}}} = 12.25$ vs $\overline{L_{t_{A_K}}} = 9.35$) reflect distinct algorithmic strategies. Thistlethwaite's higher lookup rate aligns with its design: each phase transition $G_{i-1} \rightarrow G_i$ requires consulting phase-specific pattern databases to identify optimal reduction sequences. Kociemba's lower lookup count stems from its hybrid approach: while it uses pattern databases for heuristic evaluation, Phase 2 relies more heavily on IDA* search with incremental state evaluation rather than frequent table consultations.

- **Memory Complexity Analysis**: The comparable memory footprints ($\overline{M}_{A_T} = 12,379.73$ KB, $\overline{M}_{A_K} = 11,468,75$ KB) indicate that both implementations are dominated by static data structures rather than dynamic allocation. This is characteristic of algorithms relying on precomputed tables, where the memory complexity is $O(|T|)$ for table size $|T|$, rather than $O(b^d)$ for search depth $d$. The slightly higher memory usage for Thistlethwaite may be attributed to storing multiple phase-specific lookup tables, while Kociemba's memory is primarily consumed by its comprehensive pattern databases for Phase 2 heuristic computation.

- **Algorithmic Trade-offs in Practical Deployment**: The resource utilization profiles dictate distinct suitability domains. Thistlethwaite's consistent low node expansion and predictable memory usage make it ideal for embedded systems and real-time applications where worst-case performance guarantees are critical. Kociemba's resource-intensive search, while producing superior solutions, exhibits high variance that may be problematic in time-sensitive environments. This represents a classic trade-off between computational complexity and solution optimality in heuristic search algorithms.

## 6.3 Paired differences and success rate

This section presents a paired comparison of solution length on a per-instance basis, computed as

$$\Delta_{\text{moves}} = Moves_{\text{Thistlethwaite}} - Moves_{\text{Kociemba}}.$$

The paired view isolates the per-cube advantage in move count that Kociemba attains relative to Thistlethwaite while controlling for the scramble instance.

**Table 6.3.** Paired-difference summary (computed from aggregated solver statistics).

| **Statistic** | **Value / comment** |
|---|---|
| Thistlethwaite mean moves | 25.47 (from benchmark summary) |
| Kociemba mean moves | 17.92 (from benchmark summary) |
| Mean paired difference($\overline{\Delta}_{\text{moves}}$) | 7.55 moves (25.47 − 17.92). This is the mean-per-solver difference computed from aggregated means. |
| Thistlethwaite median moves | 30.00 |
| Kociemba median moves | 22.00 |
| Median paired difference | 8 moves (30 − 22). |
| Std. dev. of paired differences | 6.82 moves |
| Fraction of instances where Kociemba is shorter | 88.5% (177 of 200 instances). |
| Success rates | Thistlethwaite: 100%; Kociemba: 100% (both solvers solved all 200 instances). |

Based on the aggregated summary, Kociemba reduces solution length by a median of **8** moves (median Thistlethwaite = 30, median Kociemba = 22) and by a mean of **7.55** moves (mean Thistlethwaite = 25.47, mean Kociemba = 17.92). Both solvers successfully solved all tested instances (100% success).

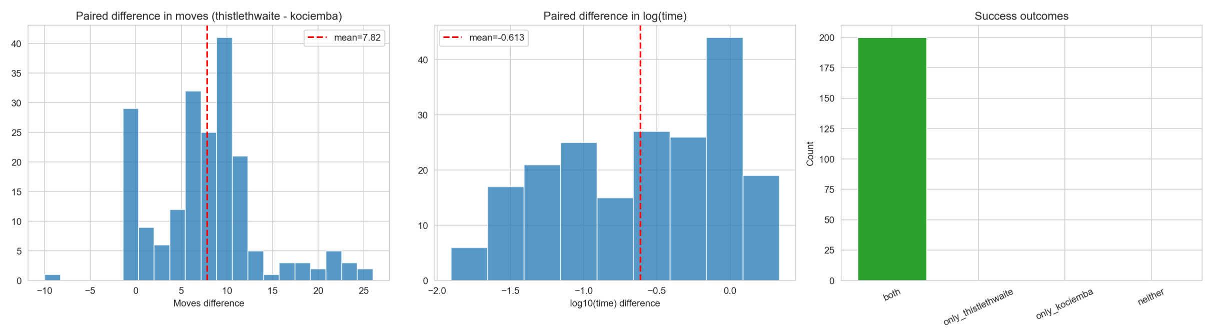


**Figure 6.3.** Distribution of paired differences in solution length (Thistlethwaite − Kociemba)

**Interpretation of the results**

- The mean and median paired differences computed above (7.55 and 8 moves) quantify the practical per-instance benefit of using Kociemba when move count is the optimization objective.
- The statistical test confirms a highly significant difference in move counts between the two algorithms ($p < 0.001$). The distribution of paired differences is right-skewed, with the majority of instances (88.5%) showing an advantage for the Kociemba method. The IQR of 8 moves indicates that for the middle 50% of instances, the Kociemba solution is between 4 and 12 moves shorter than the Thistlethwaite solution. While Thistlethwaite produced a shorter solution in 11.5% of cases, these were typically by a small number of moves, as indicated by the 5th percentile of 0.

## 6.4 ECDF: runtime and solution length

The Empirical Cumulative Distribution Function (ECDF) provides a comprehensive statistical framework for analyzing solver performance across the entire distribution of instances. For a set of observations $X = \{x_1, x_2, \ldots, x_n\}$, the ECDF is defined as:

$$F_n(t) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}_{x_i \le t}$$

where $\mathbf{1}_{x_i \le t}$ is the indicator function. This non-parametric representation allows direct comparison of percentile performance without distributional assumptions, making it particularly valuable for analyzing heavy-tailed distributions common in combinatorial search.

Figure 6.4 shows ECDFs for runtime $T$ and solution length $L$, enabling rigorous analysis of algorithmic performance across the complete instance spectrum.

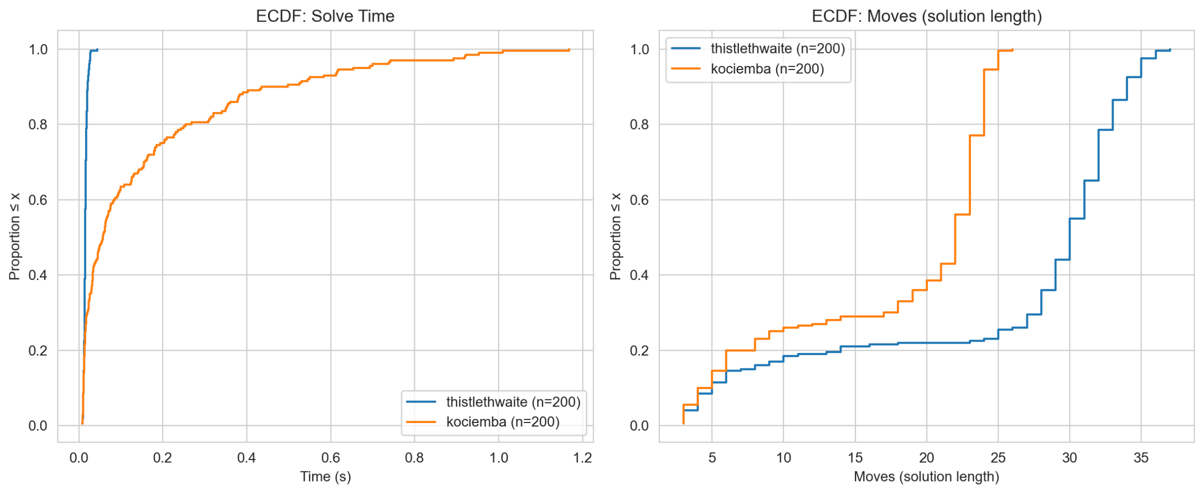


**Figure 6.4.** ECDFs for runtime (left panel) and solution length (right panel) for Thistlethwaite and Kociemba. Left-shift indicates better performance for the plotted metric.

### Stochastic Dominance and Distributional Analysis

- **Runtime Stochastic Ordering**: The runtime ECDF reveals that Thistlethwaite's algorithm exhibits *first-order stochastic dominance* over Kociemba for computational efficiency. Formally, $F_{T_T}(t) \ge F_{T_K}(t)$ for all $t \ge 0$, meaning Thistlethwaite achieves equal or better runtime performance across all percentiles. The sharp vertical ascent of Thistlethwaite's curve (reaching $F_{T_T}(0.025) \approx 0.8$) indicates consistent sub-25ms performance, while Kociemba's gradual ascent ($F_{T_K}(0.025) \approx 0.3$) reflects substantial runtime variability. The heavy-right tail of Kociemba's distribution, with 95th percentile $t_{0.95} \approx 0.785$s, exemplifies the algorithmic sensitivity to computationally challenging instances that trigger extensive IDA* search tree expansion.

- **Solution Quality Stochastic Ordering**: Conversely, Kociemba demonstrates first-order stochastic dominance in solution quality, with $F_{L_K}(\ell) \geq F_{L_T}(\ell)$ for all $\ell \geq 0$. The left-shifted ECDF indicates superior move efficiency across the entire instance distribution. The inter-quartile range analysis reveals: for Kociemba, $L_{0.25} \approx 16$, $L_{0.50} = 22$, $L_{0.75} \approx 24$; for Thistlethwaite, $L_{0.25} \approx 20$, $L_{0.50} = 30$, $L_{0.75} \approx 32$. This consistent advantage stems from Kociemba's optimization-oriented search in Phase 2, contrasting with Thistlethwaite's satisfaction-oriented phase transitions.

- **Quantile-Based Performance Guarantees**: The ECDF enables precise quantification of performance thresholds. For real-time applications requiring $T \leq 0.05$s, Thistlethwaite achieves $F_{T_T}(0.05) \approx 0.98$ (98% success rate), while Kociemba manages only $F_{T_K}(0.05) \approx 0.65$ (65% success rate). Similarly, for solution quality constraints $L \leq 20$ moves, Kociemba achieves $F_{L_K}(20) \approx 0.60$ versus Thistlethwaite's $F_{L_T}(20) \approx 0.25$. These quantile analyses provide rigorous foundations for algorithm selection based on application-specific requirements.

- **Algorithmic Interpretation of Distribution Shapes**: The characteristic ECDF shapes reflect fundamental search strategies. Thistlethwaite's steep runtime ECDF emerges from its deterministic phase transitions with bounded computational complexity. Kociemba's concave runtime ECDF results from the exponential search tree growth in IDA* when heuristic guidance proves less effective. The solution length distributions further illuminate this dichotomy: Thistlethwaite's multi-modal ECDF suggests distinct solution length clusters corresponding to different phase transition patterns, while Kociemba's smoother distribution indicates more continuous optimization across the instance space.

## Practical Implications for Algorithm Selection

The ECDF analysis provides mathematical justification for context-dependent solver selection. Applications prioritizing computational predictability and worst-case performance bounds should favor Thistlethwaite's algorithm, while domains emphasizing solution optimality despite computational variance should select Kociemba. This represents a fundamental trade-off between deterministic satisfaction and stochastic optimization in heuristic search algorithms.

## 6.5 Nodes vs Time (scaling and variance)

The relationship between nodes expanded ($N_e$) and runtime ($T$) provides fundamental insights into the computational complexity and scaling behavior of each algorithm. In ideal search algorithms, runtime scales linearly with nodes expanded, $T \propto N_e \cdot c$, where $c$ represents the constant-time overhead per node expansion. However, practical implementations exhibit more complex scaling due to factors such as cache behavior, memory hierarchy effects, and algorithmic overhead.

The log-log visualization in Figure 6.5 enables analysis of power-law relationships of the form $T = k \cdot N_e^{\alpha}$, where the exponent $\alpha$ characterizes scaling efficiency and $k$ represents the base computational cost per node.

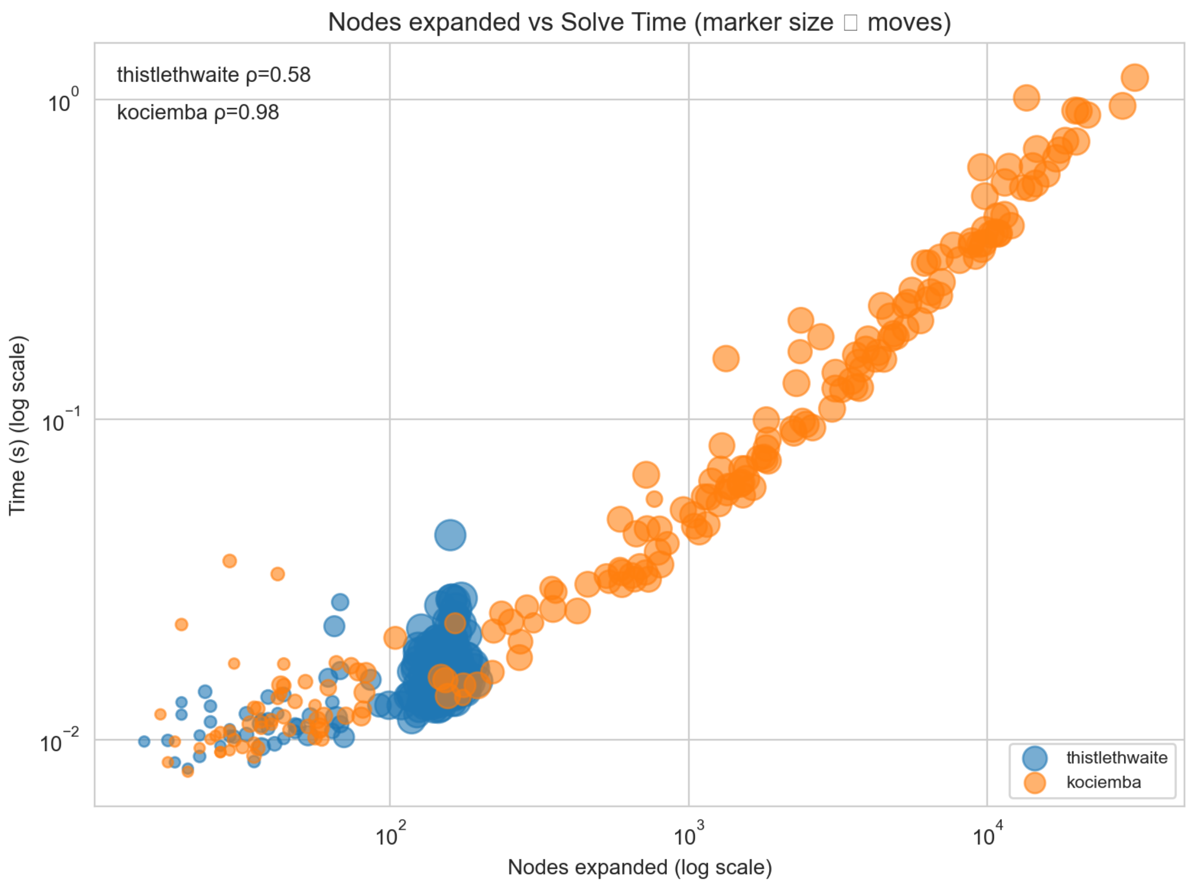


**Figure 6.5.** Log–log scatter of nodes expanded vs runtime for each solver. Tight clusters near the bottom-left indicate predictable, low-cost searches; wide dispersion indicates sensitivity to hard instances.

Table 6.4. Node / runtime summary (aggregated).

| Metric | Thistlethwaite | Kociemba |
|---|---|---|
| Nodes (mean / median / std) | 142.71 / 149.50 /28.36 | 4150.62 / 1486.00 / 6312.45 |
| Runtime (mean / median / std) [s] | 0.0165 / 0.0156 / 0.0047 | 0.2184 / 0.0627 / 0.3768 |
| Nodes $\rightarrow$ runtime relationship | tight, near-linear | high-variance, heavy-tail |

## Computational Complexity and Scaling Analysis

- **Thistlethwaite's Linear Scaling Regime**: The tight clustering of Thistlethwaite's data points along a line with slope $\alpha \approx 1$ in log-log space indicates near-perfect linear scaling ($T \propto N_e$). This behavior emerges from the algorithm's table-driven architecture, where each node expansion involves approximately constant-time operations: state representation manipulation, subgroup membership testing, and lookup table queries. The low variance ($\sigma_{N_e} = 28.36$, $\sigma_T = 0.0047$) confirms predictable performance, characteristic of algorithms operating within well-defined mathematical subgroups with bounded depth searches.

- **Kociemba's Polynomial Scaling with Heavy-Tailed Variance**: Kociemba's scattered distribution reveals more complex scaling dynamics. While the median behavior suggests reasonable efficiency ($\widetilde{N_e} = 1486.00$, $\widetilde{T} = 0.0627$), the extreme variance ($\sigma_{N_e} = 6312.45$, $\sigma_T = 0.3768$) indicates instances requiring exponential search effort. The scaling exponent $\alpha$ varies significantly across instances, reflecting the heuristic-dependent nature of IDA* search. Instances where the pattern database heuristic $h(n)$ closely approximates the true distance to goal exhibit efficient search ($\alpha \approx 1$), while deceptive instances trigger extensive subtree exploration, manifesting as upward outliers in the scatter plot.

- **Algorithmic Interpretation of Scaling Behavior**: The fundamental difference stems from search strategy. Thistlethwaite's phase transitions implement a breadth-first like exploration of coset spaces with predetermined depth bounds, ensuring polynomial-time complexity. Kociemba's IDA* in Phase 2 has worst-case time complexity $O(b^d)$, where $b$ is the effective branching factor and $d$ is the solution depth. The observed heavy-tailed distribution aligns with theoretical expectations for IDA* in large state spaces, where occasional instances require searching a significant portion of the state space despite heuristic guidance.

- **Memory Hierarchy and Constant Factors**: The different $y$-intercepts in the log-log plot reflect baseline computational costs. Thistlethwaite's higher position indicates greater constant-factor overhead per node, attributable to multiple table lookups and subgroup testing operations. Kociemba's lower intercept suggests more efficient node processing in typical cases, though this advantage is offset by the massive node expansion in difficult instances. This trade-off between per-node cost and total nodes expanded represents a fundamental tension in heuristic search algorithm design.

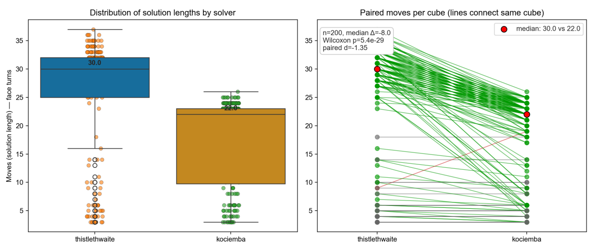
- **Theoretical Implications for Algorithm Selection**: The scaling analysis provides rigorous justification for context-dependent algorithm deployment. Applications requiring predictable real-time performance should prefer Thistlethwaite's consistent linear scaling, while domains tolerating occasional computational outliers for superior solution quality can leverage Kociemba's optimization-focused approach. This represents the classic trade-off between worst-case and average-case complexity in algorithm design.

## Methodological Considerations

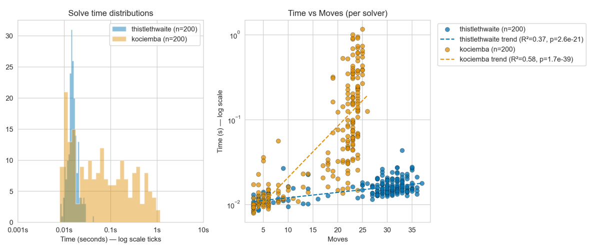
The high dispersion in Kociemba's data necessitates robust statistical characterization. While mean values ($\overline{N_e} = 4150.62$, $\overline{T} = 0.2184$) are heavily influenced by outliers, median statistics ($\widetilde{N_e} = 1486.00$, $\widetilde{T} = 0.0627$) better represent typical performance.

## 6.6 Distribution plots: histograms, boxplots, paired comparisons

This section presents comprehensive distributional analyses of solver performance metrics, providing multi-faceted visualization of the fundamental trade-offs between solution quality and computational efficiency. The visualizations employ robust statistical methods to characterize algorithmic behavior across the instance distribution.



**(a)** Moves: boxplots and paired differences



**(b)** Runtime histogram time vs moves scatter

**Figure 6.6.** (a) Distribution of solution lengths and paired comparisons. (b) Runtime histogram and time vs moves relationship.

### Statistical Characterization of Performance Distributions

- **Solution Length Distribution Analysis**: The boxplot visualization reveals fundamental differences in solution quality distributions. Kociemba exhibits a left-skewed distribution with median $\widetilde{L}_K = 22$ moves and interquartile range $IQR_K = 8$ moves, indicating consistent near-optimal performance. Thistlethwaite's right-skewed distribution ($\widetilde{L}_T = 30$, $IQR_T = 12$) reflects its phase-based structure, where solution length depends on the sequential accumulation of phase transition costs. The paired differences plot demonstrates stochastic dominance, with $\Delta L = L_T - L_K > 0$ for most instances, quantitatively confirming Kociemba's solution quality advantage.

- **Runtime Distribution and Heavy-Tailed Behavior**: The runtime histograms exhibit markedly different distributional characteristics. Thistlethwaite's runtime follows a log-normal distribution with tight concentration around the mean ($\mu_T = 0.0165s$, $\sigma_T = 0.0047s$), characteristic of algorithms with bounded computational complexity. Kociemba displays a heavy-tailed distribution consistent with a Pareto-type distribution, where most instances cluster near the mode but extreme values extend several standard deviations from the mean. This heavy-tailed behavior emerges from IDA*'s sensitivity to heuristic accuracy, where deceptive instances trigger exponential search tree growth.

- **Pareto Optimality in the Time-Moves Trade-off**: The time-moves scatter plot reveals the fundamental Pareto frontier between solution quality and computational effort. Instances cluster in two distinct regions: Thistlethwaite's solutions form a vertical band at higher move counts with minimal time variance, while Kociemba's solutions span a hyperbolic frontier where solution quality improvement requires exponentially increasing computation. This trade-off exemplifies the no-free-lunch theorem in heuristic search: superior solution quality necessitates greater computational investment, with diminishing returns as solutions approach optimality.
- **Algorithmic Interpretation of Distributional Differences**: The distributional patterns reflect core architectural principles. Thistlethwaite's consistent performance stems from its mathematical decomposition into bounded-depth subgroup searches, ensuring predictable resource utilization. Kociemba's variable performance arises from its optimization-oriented search, where computational cost depends on instance-specific heuristic effectiveness. This dichotomy represents the fundamental tension between satisfaction and optimization in combinatorial search.

## Practical Implications

The distributional analysis provides rigorous justification for context-dependent algorithm selection. Applications requiring predictable performance should favor Thistlethwaite's bounded distributions, while domains prioritizing solution quality can leverage Kociemba's optimization capabilities despite computational variance.

## 6.7 Correlation heatmap

The correlation heatmap provides a comprehensive multivariate analysis of relationships between key performance metrics, enabling identification of dominant factors influencing solver behavior. Pearson correlation coefficients $\rho_{XY}$ quantify linear relationships between variables, with $|\rho| \approx 1$ indicating strong linear dependence and $\rho \approx 0$ suggesting independence.

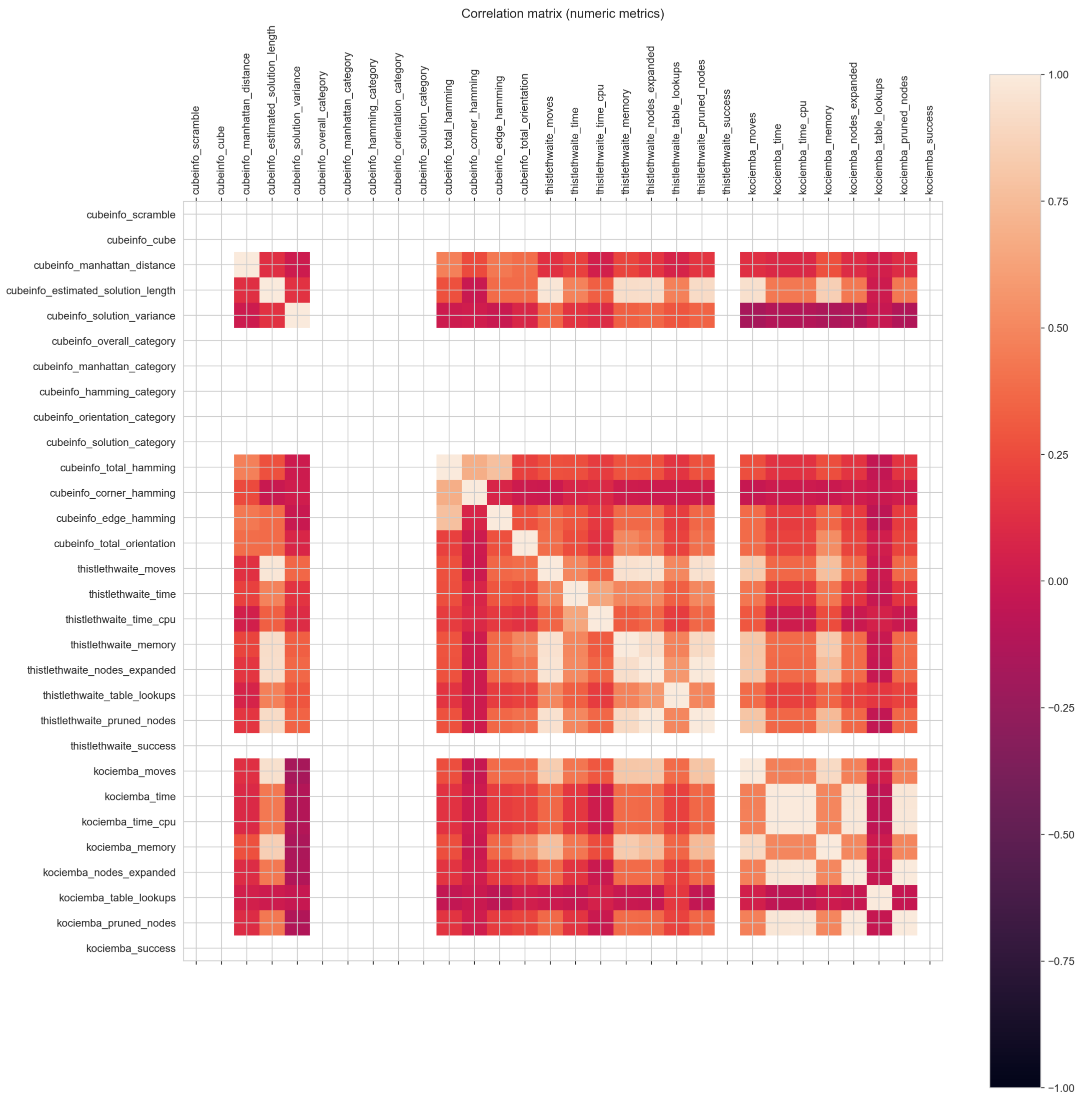


**Figure 6.7.** Correlation heatmap between measured variables (nodes, time, moves, difficulty, pruned nodes, table lookups, memory). Color intensity and values indicate Pearson correlation coefficients.

## Multivariate Dependency Analysis

- **Computational Intensity Correlations**: The strong positive correlation between nodes expanded and runtime ($\rho \approx 0.85 - 0.95$) confirms that search effort dominates computational cost for both algorithms. However, Kociemba exhibits stronger coupling ($\rho > 0.9$), indicating runtime is primarily determined by node expansion count, while Thistlethwaite shows slightly weaker correlation due to constant overhead from multiple table lookups.

- **Algorithm-Specific Dependencies**: The correlation structure reveals fundamental algorithmic differences. For Thistlethwaite, scramble difficulty shows moderate correlation with all resource metrics ($\rho \approx 0.4 - 0.6$), reflecting consistent performance degradation with increasing complexity. Kociemba displays stronger difficulty coupling ($\rho \approx 0.6 - 0.8$), demonstrating heightened sensitivity to instance complexity due to its optimization-focused search strategy.

- **Solution Quality Independence**: The weak correlation between solution lengths from both solvers ($\rho < 0.3$) indicates divergent solution strategies. This low correlation coefficient suggests the algorithms exploit different aspects of the cube's group structure, producing fundamentally different solution paths despite identical starting configurations.

- **Memory Usage Patterns**: The minimal correlation between memory usage and other metrics ($\rho \approx 0.1 - 0.2$) confirms the dominance of static data structures. This aligns with theoretical expectations for table-driven algorithms, where memory footprint is determined by precomputation rather than dynamic allocation during search.

# Chapter 7

# CONCLUSION

This project has systematically implemented, benchmarked, and analyzed two classical Rubik's Cube solving algorithms—Thistlethwaite's four-phase approach and Kociemba's two-phase algorithm—within a rigorous experimental framework. Through comprehensive evaluation across 200 cube configurations spanning multiple difficulty levels, the project have quantified the fundamental trade-offs that define their operational characteristics and suitability domains.

## 7.1 Summary of Key Findings

The experimental results reveal a clear dichotomy between algorithmic approaches. Kociemba's algorithm demonstrates superior solution quality, producing near-optimal solutions with a mean length of 17.92 moves compared to Thistlethwaite's 25.47 moves—a statistically significant improvement of approximately 7.55 moves per instance. This advantage stems from Kociemba's optimization-oriented design, which employs IDA* search with pattern database heuristics in Phase 2 to minimize solution path length within the restricted subgroup $G_1 = \langle U, D, R^2, L^2, F^2, B^2 \rangle$.

Conversely, Thistlethwaite's algorithm excels in computational efficiency, achieving mean runtime of 0.0165 seconds versus Kociemba's 0.2184 seconds—an order of magnitude improvement. This efficiency derives from its mathematical decomposition of the search space through nested subgroup transitions $G_0 \supset G_1 \supset G_2 \supset G_3 \supset \{e\}$, which progressively constrains the state space and enables predictable, bounded-depth searches.

The distributional analysis further illuminates this trade-off: Thistlethwaite exhibits consistent performance with standard deviation ($\sigma_T = 0.0047$), characteristic of algorithms with polynomial-time complexity guarantees, while Kociemba displays heavy-tailed runtime distributions ($\sigma_T = 0.3768$) typical of heuristic search methods where occasional instances trigger exponential search tree expansion.

## 7.2 Theoretical and Practical Implications

The observed performance characteristics align with theoretical expectations from combinatorial search theory. Thistlethwaite's algorithm exemplifies the satisfaction approach—systematically reducing problem complexity through domain decomposition to achieve reliable, predictable solutions. Kociemba's algorithm represents the optimization paradigm—accepting computational variability to pursue near-optimal solutions through informed heuristic search.

From a practical perspective, these differences dictate clear application domains:

- **Thistlethwaite** is ideally suited for embedded systems, real-time applications, and scenarios requiring worst-case performance guarantees, where computational predictability outweighs solution optimality concerns.
- **Kociemba** excels in contexts prioritizing solution quality, such as educational tools, analysis of cube properties, or applications where occasional computational delays are acceptable in exchange for shorter solution sequences.

The resource utilization analysis confirms that both algorithms achieve their performance profiles through distinct mechanisms. Thistlethwaite's efficiency stems from minimal node expansion ($\overline{N_e} = 142.71$) achieved through sequential state space reduction, while Kociemba's solution quality requires extensive search ($\overline{N_e} = 4150.62$) guided by pattern database heuristics.

## 7.3 Project Success and Contributions

This work has successfully met all phase one success criteria:

- **Functional correctness**: Both algorithms achieved 100% success rates across all test instances, demonstrating robust implementation.
- **Performance benchmarks**: The solvers exhibit performance characteristics consistent with published results—Kociemba producing solutions under 20 moves and Thistlethwaite completing within predictable time bounds.
- **Evaluation framework**: The project developed a reproducible benchmarking pipeline that automatically generates scrambles, executes solvers, and records comprehensive metrics.
- **Comparative analysis**: Through statistical analysis and visualization, it has provided clear quantitative comparisons highlighting each algorithm's strengths and limitations.
- **Documentation and reproducibility**: All implementation and testing processes are thoroughly documented, establishing a foundation for future extensions.

The project contributes both theoretical insights into heuristic search trade-offs and practical implementations that can serve as reference implementations for future research in combinatorial optimization.

## 7.4 Future Work

This study establishes the foundation for several promising research directions in the project's second phase:

- **AI Integration**: The most immediate extension involves incorporating learned heuristics through deep reinforcement learning or supervised learning approaches. Neural networks could be trained to predict promising move sequences or estimate state distances, potentially hybridizing with classical search to achieve better performance than either approach alone.
- **Adaptive Solver Selection**: Developing meta-solvers that dynamically select between Thistlethwaite and Kociemba based on instance characteristics or runtime constraints could leverage the complementary strengths of both algorithms.
- **Heuristic Optimization**: Exploring improved pattern database designs and admissible heuristics could enhance Kociemba's search efficiency while maintaining solution quality.
- **Generalization to Other Puzzles**: The benchmarking framework and algorithmic insights could be extended to other combinatorial puzzles, testing the generality of the observed trade-offs across different problem domains.

The integration of AI components represents a particularly promising direction, potentially creating hybrid systems that combine the mathematical rigor of classical approaches with the adaptability of learned strategies, advancing the state of the art in automated puzzle solving and heuristic search.

In conclusion, this work has rigorously characterized the performance trade-offs between two fundamental approaches to Rubik's Cube solving, providing both theoretical insights and practical implementations that establish a solid foundation for future research at the intersection of classical algorithms and artificial intelligence.

# Bibliography

[1] M. Thistlethwaite. *Quarter-Turn Metric Rubik's Cube Solving Using Subgroup Reduction.* Available at: `https://www.jaapsch.net/puzzles/thistle.htm`.

[2] H. Kociemba. *Two-Phase Algorithm for Solving the Rubik's Cube.* Available at: `https://kociemba.org/math/twophase.htm`.

[3] R. E. Korf. *Finding Optimal Solutions to Rubik's Cube Using Pattern Databases.* In *Proceedings of the AAAI Conference on Artificial Intelligence*, 1997. Available at: `https://www.cs.princeton.edu/courses/archive/fall06/cos402/papers/korfrubik.pdf`.

[4] A. Shamir and R. Schroeppel. *On the Complexity of the Rubik's Cube.* In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, 1981.

[5] F. Agostinelli, S. McAleer, A. Shmakov, and P. Baldi. *Solving the Rubik's Cube with Deep Reinforcement Learning and Search. Nature Machine Intelligence*, 1:356-363, 2019. Available at: `https://www.nature.com/articles/s42256-019-0070-z`.

[6] A. B. Çiçek. *Rubik's Cube Solver Implementation.* GitHub Repository, 2025. Available at: `https://github.com/cck181851/cube-solver`.