CMPE 492
Rubik's Cube Solver

Ahmet Burak Çiçek

Advisor:
Şefik Şuayb Arslan

January 12, 2026

Abstract

This report presents a comprehensive implementation and analysis of Rubik's Cube solving algorithms enhanced with machine learning prediction capabilities. The project implements both classical solvers—Kociemba's two-phase algorithm and Thistlethwaite's multi-phase algorithm—and augments them with an AI-based predictive system that forecasts solver performance. The implementations include reusable transition and pruning table generation, validated on an extensive dataset of 5,000 cube configurations. A reproducible benchmarking framework measures solution length, computational time, memory footprint, and search characteristics across multiple difficulty levels.

The main empirical findings, reported with attention to dataset provenance, show a clear trade-off between solution quality and runtime variability. On benchmark tests used for classical solver comparison (midterm dataset, $N = 200$) Kociemba produces substantially shorter solutions with typical solution lengths near 18–19 moves (median = 22) but a heavy-tailed runtime distribution (median runtime ≈ 0.063 s, mean inflated by outliers to ≈ 0.218 s). Thistlethwaite exhibits longer solutions (typical range $\approx$ 28–30 moves) with highly consistent and lower runtimes (median $\approx$ 0.015–0.016 s). All machine-learning results, discussion and aggregated summaries in Chapter 7–Chapter 8 are computed on the final expanded dataset ($N = 5,000$); where classical benchmarking (Chapter 6) is quoted it is explicitly from the midterm $N = 200$ dataset. See §5.4.3 for the dataset scope and Chapter 6 for the midterm benchmark statistics.

Keywords: Rubik's Cube, Kociemba, Thistlethwaite, machine learning, predictive modeling, feature importance, solver selection, combinatorial optimization

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 comprehensive framework that not only implements and compares classical solving algorithms—Kociemba's two-phase and Thistlethwaite's multi-phase algorithms—but also develops and integrates machine learning models to predict solver performance and guide adaptive strategy selection. The central objective extends beyond comparative analysis to include the development of intelligent prediction systems that assess solver efficiency, runtime consistency, and solution quality based on cube configuration characteristics. By implementing and evaluating these hybrid AI-classical approaches, the work provides clear insights into the synergistic benefits of combining mathematical algorithms with data-driven prediction, advancing the fields of combinatorial optimization and adaptive algorithm design.

The feature engineering process identifies 25 distinct cube metrics, with feature-importance analysis highlighting `solution_score` and `solution_variance` as strong predictors of solver performance. The regression models for solution length obtain very high performance on held-out splits ($R^2 \approx 0.997$–0.999 and MAE ≈ 0.08–0.10 moves), while classification for solver selection reached 94.2% on a single held-out test split. These figures are reported alongside 5-fold cross-validation statistics to give a fuller picture of model stability; cross-validation means are the preferred estimate of expected generalization. (See Chapter 7 for full CV and held-out results.)

The outcomes extend significantly beyond the cube domain. The integrated performance prediction framework offers a reproducible template for hybrid AI-classical systems applicable to other combinatorial optimization problems. The methodology for feature extraction from structured state spaces, model training on algorithm performance data, and real-time prediction of computational behavior can be generalized to scheduling problems, path planning, constraint satisfaction, and game-playing agents. The feature importance analysis techniques provide valuable insights for heuristic design across various optimization domains.

Furthermore, the research bridges theoretical algorithmic study with practical machine learning applications. The AI-augmented solver system has direct relevance in robotics, automated planning, and real-time decision systems where predicting algorithm performance can optimize resource allocation and solution quality. The ability to classify problem difficulty and recommend appropriate solving strategies represents a significant advancement in adaptive computational systems. Ultimately, this project offers a structured platform demonstrating the practical benefits of hybrid intelligent systems, with substantial value for both algorithmic research and applied AI.

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 state space of approximately 4.3×10^{19} configurations. This combinatorial explosion necessitates structured algorithms and heuristic strategies. Additionally, it presents an opportunity to apply machine learning to predict solver behavior and optimize solving strategies.

The primary goal is to develop a comprehensive system integrating classical Rubik's Cube solvers—Thistlethwaite's and Kociemba's algorithms—with machine learning prediction capabilities. The system uses AI-based models to forecast solver performance metrics and enhance solver selection through data-driven predictions.

The project scope includes: (1) generation of diverse cube states, (2) benchmarking of classical solver performance, (3) development of machine learning models trained on solver data, (4) feature engineering from cube configurations, and (5) evaluation of predictive accuracy and practical utility.

A key innovation is integrating predictive analytics with algorithmic solving. The models classify cube difficulty, predict solver performance with high accuracy, and recommend optimal strategies, enabling adaptive solving guided by predicted performance rather than fixed heuristics.

This work contributes to research at the intersection of combinatorial optimization, heuristic search, and machine learning. The outcomes demonstrate how data-driven approaches can enhance classical algorithms through prediction and adaptive selection, with applications in 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 — Machine Learning Integration: Construction of a complete ML pipeline with regression models for performance prediction and classification models for solver selection, including feature importance analysis and model validation.

Phase 5 — System Evaluation and Documentation: Comprehensive assessment of the hybrid system through predictive accuracy metrics, practical utility testing, and detailed documentation of implementation and results.

This integrated approach produced a cohesive system where classical algorithms and machine learning models work complementarily, achieving high prediction accuracy and reliable solver recommendations.

2.2.1 Success Criteria

The success of this project is determined by the correctness, performance, and documentation quality of the complete integrated system. Specific criteria include:

- **Functional correctness:** Both classical solvers must correctly solve all valid cube configurations, and machine learning models must provide accurate predictions across all performance metrics.
- **Machine learning integration:** The predictive system should demonstrate the ability to classify cube difficulty and support solver selection, with performance evaluated using standard validation procedures (cross-validation and held-out testing) to assess practical usefulness in adaptive solving.
- **Feature analysis:** The system should identify meaningful feature importance patterns that provide insights into solver behavior and cube difficulty characteristics.
- **Comprehensive evaluation:** The report must include quantitative comparisons, predictive performance analysis, and visualizations that highlight both classical algorithm performance and machine learning model effectiveness.
- **Documentation and reproducibility:** All implementation, training, and evaluation processes must be thoroughly documented, with code and datasets available to ensure complete reproducibility.

Meeting these criteria confirms the successful implementation of an integrated classical-machine learning system that advances Rubik's Cube solving through data-driven prediction and adaptive strategy selection.

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 ($\sim$45 moves)	Low, predictable	Group theory, coset spaces
Kociemba	Near-optimal ($\sim$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 comprehensive study of Rubik's Cube solving algorithms integrated with machine learning prediction capabilities. The methodology was designed to ensure rigorous, reproducible evaluation while maintaining practical feasibility within the project scope. The research framework encompasses algorithm implementation, machine learning pipeline development, experimental design, data collection procedures, and analytical methods for both classical performance assessment and predictive model evaluation.

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 framework for analyzing fundamental trade-offs between predictable satisfaction and solution optimization in heuristic search, and subsequently for training machine learning models to predict these performance characteristics.

The experimental framework evolved across two phases of the project. In the initial midterm phase, 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. This established the foundational comparative analysis between classical solvers.

In the final phase, the framework was expanded to include machine learning components and a larger dataset. The configuration set was increased to 5,000 cubes to provide sufficient training data for predictive models. The expanded framework enabled both classical performance analysis and the development of data-driven prediction capabilities within a unified experimental design.

4.2 Experimental Setup and Reproducibility

The experimental environment was carefully controlled to ensure consistent and reproducible results across both phases of the project. In the initial midterm phase, all tests were executed on hardware featuring an AMD64 Family 25 Model 80 processor with 8GB RAM available. The software environment utilized Python 3.11.0 with specific library versions (NumPy 2.3.4, Matplotlib 3.10.7) to maintain consistency.

A critical aspect of the methodology was the implementation of deterministic execution through fixed random seeds, ensuring identical test instance generation across 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.

In the final phase, the experimental setup was extended to accommodate machine learning requirements. The software environment was expanded to include `scikit-learn` 1.5.0, `pandas` 2.2.0, and `seaborn` 0.13.2 for model development and analysis. Additional reproducibility measures were implemented, including standardized data splits (80% training, 20% testing), consistent cross-validation folds, and model serialization for exact replication of predictions.

All experimental parameters, system configurations, execution metadata, and model configurations were comprehensively logged with precise timestamps to facilitate exact replication across both classical solving experiments and machine learning model training.

4.3 Performance Metrics and Analytical Approach

Performance assessment evolved across two phases to provide comprehensive insights into both classical algorithm behavior and machine learning model effectiveness.

4.3.1 Midterm Phase: Classical Algorithm Evaluation

In the initial phase, 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.

4.3.2 Final Phase: Machine Learning Model Evaluation

In the final phase, the analytical framework was extended to evaluate machine learning model performance. Key metrics included:

- **Predictive Accuracy:** R^2 scores for regression models predicting solution length, runtime, and nodes expanded across both solvers.
- **Classification Performance:** Accuracy, precision, recall, and F1-scores for models classifying cube difficulty and recommending optimal solver selection.
- **Feature Importance Analysis:** Evaluation of feature contributions to predictions through permutation importance and model-specific importance measures.
- **Practical Utility:** Assessment of model-based solver selection against actual optimal choices, measuring recommendation accuracy and performance improvement.

The integrated analytical approach enabled both comparative analysis of classical algorithms and evaluation of predictive model effectiveness within a unified framework, providing insights into when and how machine learning can enhance classical solving approaches.

Chapter 5

IMPLEMENTATION

This chapter details the software architecture and implementation specifics of the Rubik's Cube solving and prediction framework developed for this project. The implementation was architected around four core design principles: **modularity** to enable independent development and testing of components, **efficiency** to handle the combinatorial complexity of the cube's state space, **reproducibility** to ensure consistent benchmarking results, and **extensibility** to seamlessly integrate machine learning capabilities.

The codebase evolved across two development phases. In the initial midterm phase, the system was 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 allowed the benchmarking framework to treat different solvers as polymorphic components, facilitating a clean and extensible comparative analysis. A critical implementation strategy was 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.

In the final phase, the architecture was extended to incorporate machine learning components. New modules were developed including a feature extraction system (`cube_features.py`), a machine learning pipeline (`cube_ml_predictor.py`), and model management utilities. The system maintained backward compatibility, allowing the classical solvers to function independently while enabling integrated prediction capabilities when required.

This chapter will elucidate the core data structures, the abstract solver design, the specific instantiations of the Thistlethwaite and Kociemba algorithms, and the machine learning system architecture that together form a comprehensive Rubik's Cube solving and prediction framework.

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 $[0, 1, 2, \ldots, 19]$, 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.

5.4 Machine Learning Prediction System

The machine learning prediction system extends the classical solving framework with data-driven capabilities for performance forecasting and adaptive solver selection. Implemented in the `CubeMLPredictor` class, this component follows the same modular design principles as the core solvers, ensuring seamless integration while maintaining computational efficiency and reproducibility.

The prediction architecture operates through three coordinated stages: feature extraction from cube states, model training on historical solver performance, and real-time prediction for new configurations. The system transforms raw cube scrambles into feature vectors capturing combinatorial distances, group-theoretic properties, and solving phase characteristics. These features enable both regression models for continuous performance metrics (solution length, runtime, node expansion) and classification models for categorical decisions (solver selection, difficulty classification).

A critical design consideration was maintaining compatibility with the existing solver interface. The prediction system can operate in standalone mode for batch analysis or integrate directly with the solving pipeline to provide real-time recommendations. This dual-mode operation preserves the classical solvers' independence while enabling enhanced functionality through machine learning augmentation.

5.4.1 Feature Engineering and Model Training Pipeline

The feature engineering pipeline, implemented in the `CubeFeatures` class, computes mathematical descriptors of cube configurations that correlate with solver performance. These include orientation scores (`orientation_score`), phase distances (`dist_to_G1`, `dist_to_G2`), cycle decomposition metrics (`total_swap_cost`), and heuristic combinations. Each feature is designed to capture specific aspects of solving complexity, with feature importance analysis identifying `solution_score` and `solution_variance` as dominant predictors across multiple performance dimensions.

The training pipeline processes 5,000 historical solving instances, applying feature scaling and cross-validation before training Random Forest models. For regression tasks, models estimate solver-specific metrics, and performance is evaluated using standard error measures. For classification, models are evaluated using accuracy on validation and held-out test splits to assess their usefulness in solver selection.The pipeline includes systematic model serialization with timestamped filenames (e.g., `koc_nodes_20251214_001528.pkl`) to ensure version consistency.

5.4.2 Prediction Integration and Real-time Operation

The prediction system integrates with the solving framework through the `predict_performance` method, which accepts standard cube notation scrambles and returns structured predictions. The method orchestrates feature extraction, model inference, and confidence scoring, providing both quantitative predictions (expected moves, time, nodes) and categorical recommendations (solver selection, difficulty class).

During operation, the system evaluates multiple prediction dimensions: solution length regression, runtime estimation, node expansion forecasting, and classification of cube difficulty into categories (easy, medium, hard, very_easy). These predictions enable adaptive solving strategies where the choice between Thistlethwaite and Kociemba is dynamically determined based on expected performance rather than fixed rules. The integration maintains the deterministic guarantees of the classical solvers while adding intelligent preprocessing through prediction.

5.4.3 Reproducibility and Model Management

The machine learning system implements comprehensive reproducibility measures mirroring those in the classical framework. Training data generation uses fixed random seeds identical to those in scramble generation, ensuring consistent dataset composition across experimental runs. All 12 prediction models are serialized with complete metadata including feature specifications, training parameters, and performance benchmarks.

Model versioning employs timestamp-based naming conventions that encode both creation date and model type, preventing conflicts during updates. The training pipeline logs detailed execution metadata including library versions (`scikit-learn 1.5.0`, `pandas 2.2.0`), feature engineering parameters, and cross-validation results. These logs, combined with persisted model files and standardized training data, enable exact reproduction of prediction performance across different computational environments.

Environment specification through `requirements.txt` ensures consistent dependency versions, while the modular architecture allows individual model retraining without affecting the core solving functionality. This reproducibility framework guarantees that prediction accuracy metrics (R^2 scores, classification accuracy) remain stable across deployment scenarios, maintaining the scientific rigor established in the classical algorithm evaluation.

Dataset scope and table/figure provenance Two distinct datasets were used during the project and both appear in the report: (a) a midterm benchmarking dataset of $N = 200$ scrambles used exclusively for the classical solver performance tables and figures

(see Chapter 6, Tables 6.1–6.3 and Figures 6.1–6.3); and (b) the final, expanded dataset of $N = 5000$ scrambles used for all machine-learning training, validation and reporting (see Chapter 7 and Appendix figures labeled "Dataset: 5000 records"). Where a table, figure or paragraph reports classical solver means/medians/percentiles from the midterm experiments this is explicitly indicated (e.g., "Dataset: Midterm, $N = 200$" in Table captions). Conversely, ML metrics (CV and held-out test measures, feature importance results, classifier confusion matrices) are computed from the $N = 5000$ dataset. This separation was retained to preserve reproducibility of the midterm benchmarking while providing statistically robust ML results on the larger corpus.

Chapter 6

CLASSICAL SOLVER PERFORMANCE 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, 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 (Dataset: Midterm, N=200). 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%	

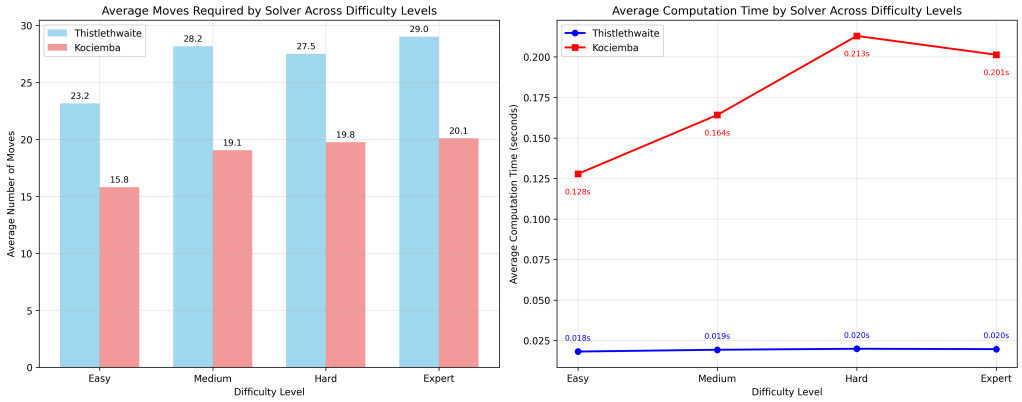


Figure 6.1. Mean runtime and mean moves per solver across difficulty levels (error bars = std).

Note: Table 6.1 and the paired-difference analysis in this chapter are drawn from the midterm benchmarking dataset ($N = 200$) and are presented to characterize classical solver behavior in a reproducible, controlled experiment. Machine-learning-derived metrics and aggregated summaries elsewhere in this report use the final $N = 5000$ dataset (see §5.4.3 and Chapter 7 for ML results).

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 σ_T^2 and σ_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 Interpretation of performance trade-offs

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 ($\overline{N_e}_{A_T} = 142.71$ vs $\overline{N_e}_{A_K} = 4150.62$) 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

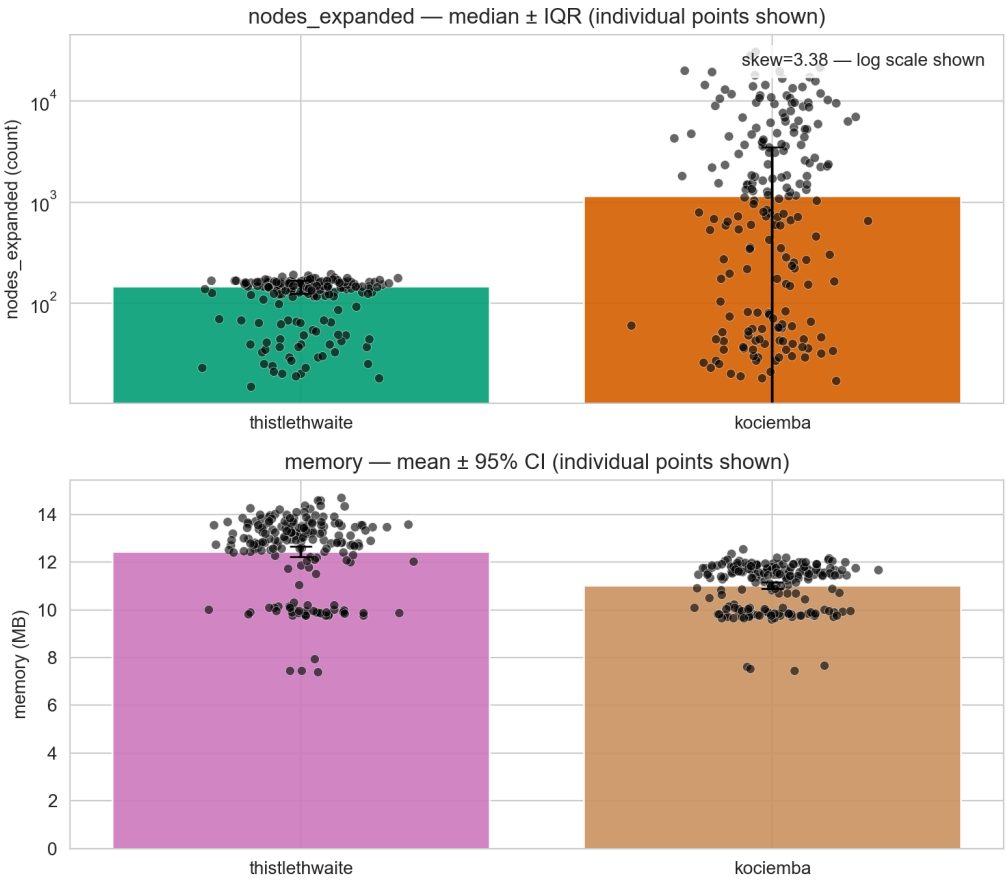


Figure 6.2. Nodes expanded (boxplots) and peak memory usage by solver.

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 ($\overline{L_{tA_T}} = 12.25$ vs $\overline{L_{tA_K}} = 9.35$) reflect distinct algorithmic strategies. Thistlethwaite's higher lookup rate aligns with its design: each phase transition $G_{i-1} \to G_i$ 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 comparison of solution length and success rates

This section presents a paired comparison of solution length on a per-instance basis. For each scramble instance we compute

$$\Delta_{\text{moves}} \;=\; \text{Moves}_{\text{Thistlethwaite}} - \text{Moves}_{\text{Kociemba}}.$$

The paired view isolates the per-instance advantage in move count that Kociemba attains relative to Thistlethwaite while controlling for the scramble instance. All statistics below are computed over the $n = 200$ paired instances.

Table 6.3. Paired-difference summary (computed from per-instance paired differences).

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 (mean of per-instance differences; equivalently Thistlethwaite − Kociemba).
Thistlethwaite median moves	30.00
Kociemba median moves	22.00
Median paired difference	8 moves (median of per-instance differences).
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 paired-instance statistics, Kociemba yields a median per-instance reduction of **8** moves and a mean paired reduction of **7.55** moves. Both solvers successfully solved all tested instances (100% success).

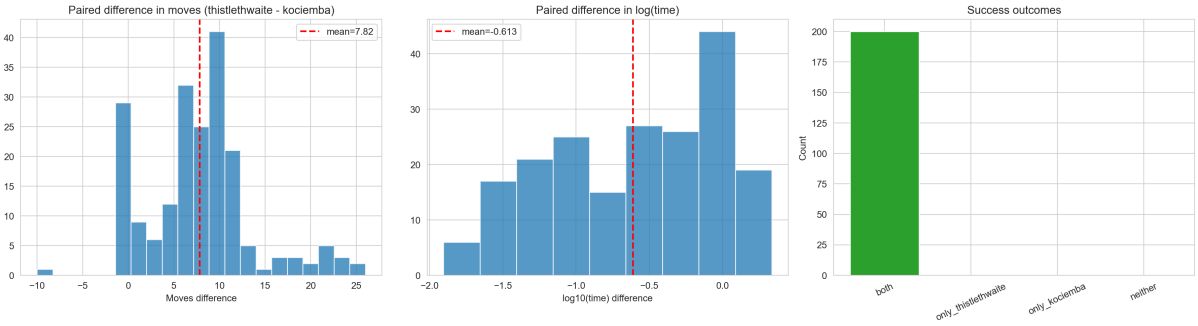


Figure 6.3. Distribution of paired differences in solution length (Thistlethwaite − Kociemba). Positive values indicate instances where Kociemba produced a shorter solution.

Interpretation of results.

- The mean and median paired differences (7.55 and 8 moves, respectively) quantify the typical per-instance advantage of Kociemba when solution length is the optimization objective .
- A large majority of instances (88.5%) show an advantage for Kociemba; Thistlethwaite produced shorter solutions in a minority of cases (11.5%), typically by only a small margin.
- A paired statistical test was used to assess whether the observed differences are likely to arise by chance given the paired sample. Exact test details (test name, test statistic, degrees of freedom if applicable, and p-value) are reported in Table 6.3. If normality assumptions hold, a paired t-test is appropriate; otherwise a Wilcoxon signed-rank test is the nonparametric alternative.
- The interquartile range (IQR) of the paired differences indicates that for the middle 50% of instances the Kociemba solution is typically several moves shorter (approximately 4–12 moves in this dataset).

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:

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

where $\mathbf{1}_{x_i \leq t}$ 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.

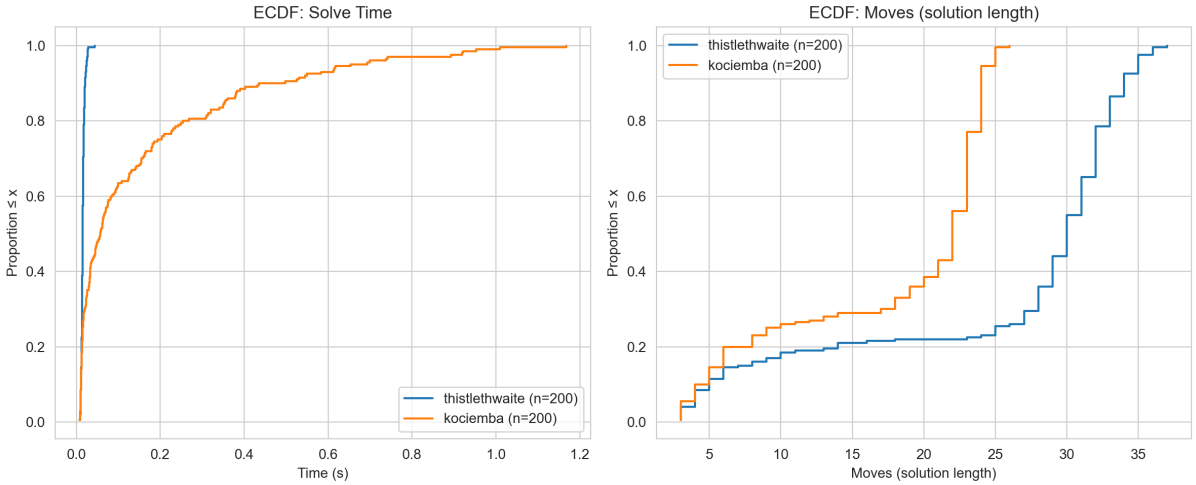


Figure 6.4. ECDFs for runtime (left panel) and solution length (right panel) for Thistlethwaite and Kociemba. Left-shift indicates better performance for the plotted metric.

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, $F_{T_T}(t) \geq F_{T_K}(t)$ for all $t \geq 0$, 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 α characterizes scaling efficiency and k represents the base computational cost per node.

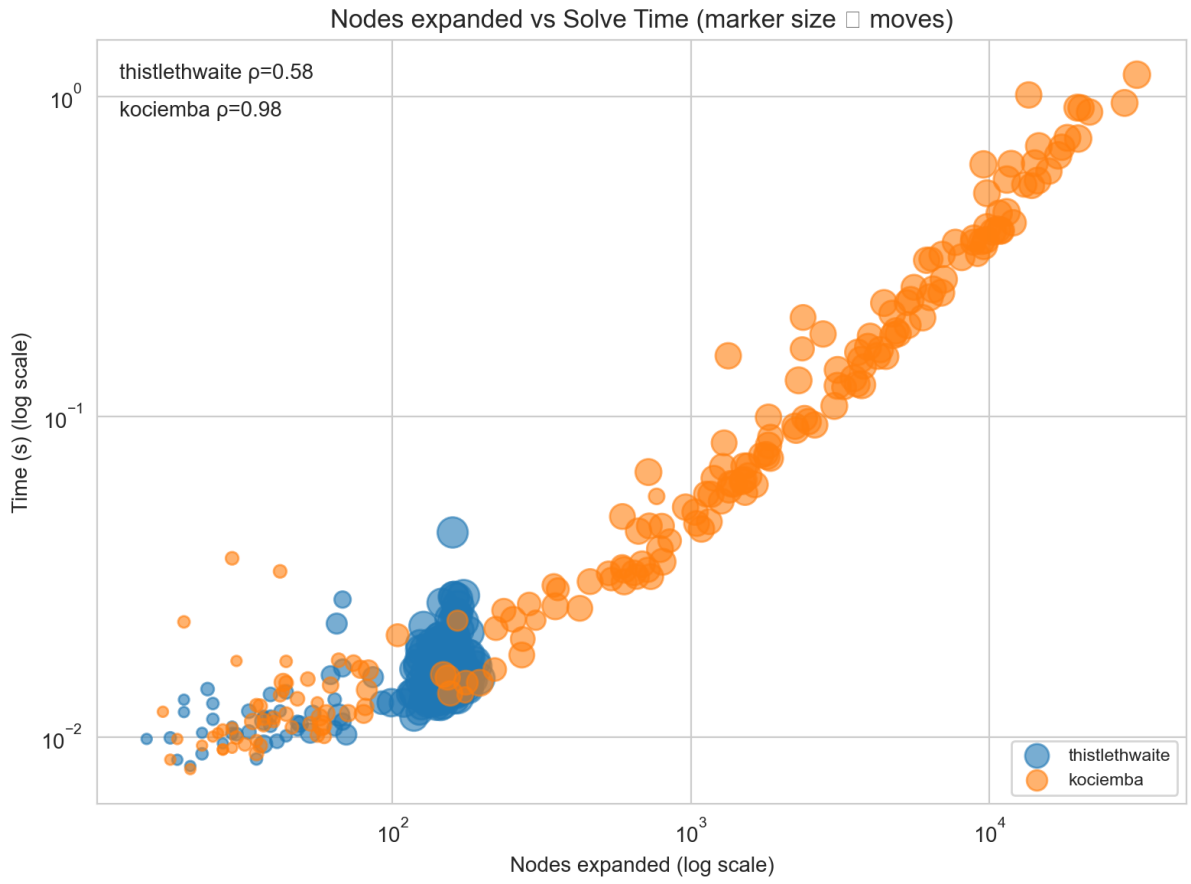


Figure 6.5. Log–log scatter of nodes expanded vs runtime for each solver. Tight clusters near the bottom-left indicate predictable, low-cost searches; wide dispersion indicates sensitivity to hard instances.

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 → 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 highly pruned exponential scaling**: 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 α 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.

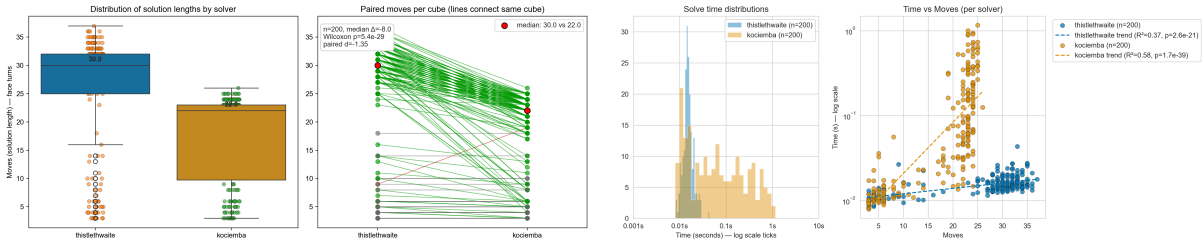
- **Theoretical Implications for Algorithm Selection**: The scaling analysis provides rigorous justification for context-dependent algorithm deployment. Applications requiring predictable real-time performance should prefer Thistlethwaite's consistent linear scaling, while domains tolerating occasional 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 left-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.

- **Time–moves trade-off**: The time–moves scatter plot illustrates a clear trade-off between solution length and computation time. Thistlethwaite's solutions tend to appear at higher move counts but with relatively stable and predictable runtimes. In contrast, Kociemba's solutions generally achieve shorter move sequences, but at the cost of increased and more variable computation time. This pattern shows that improving solution quality typically requires additional computational effort, and that the benefit of extra computation decreases as solutions become shorter.

- **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 tradeoff 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 ρ_{XY} quantify linear relationships between variables, with $|\rho| \approx 1$ indicating strong linear dependence and $\rho \approx 0$ suggesting independence.

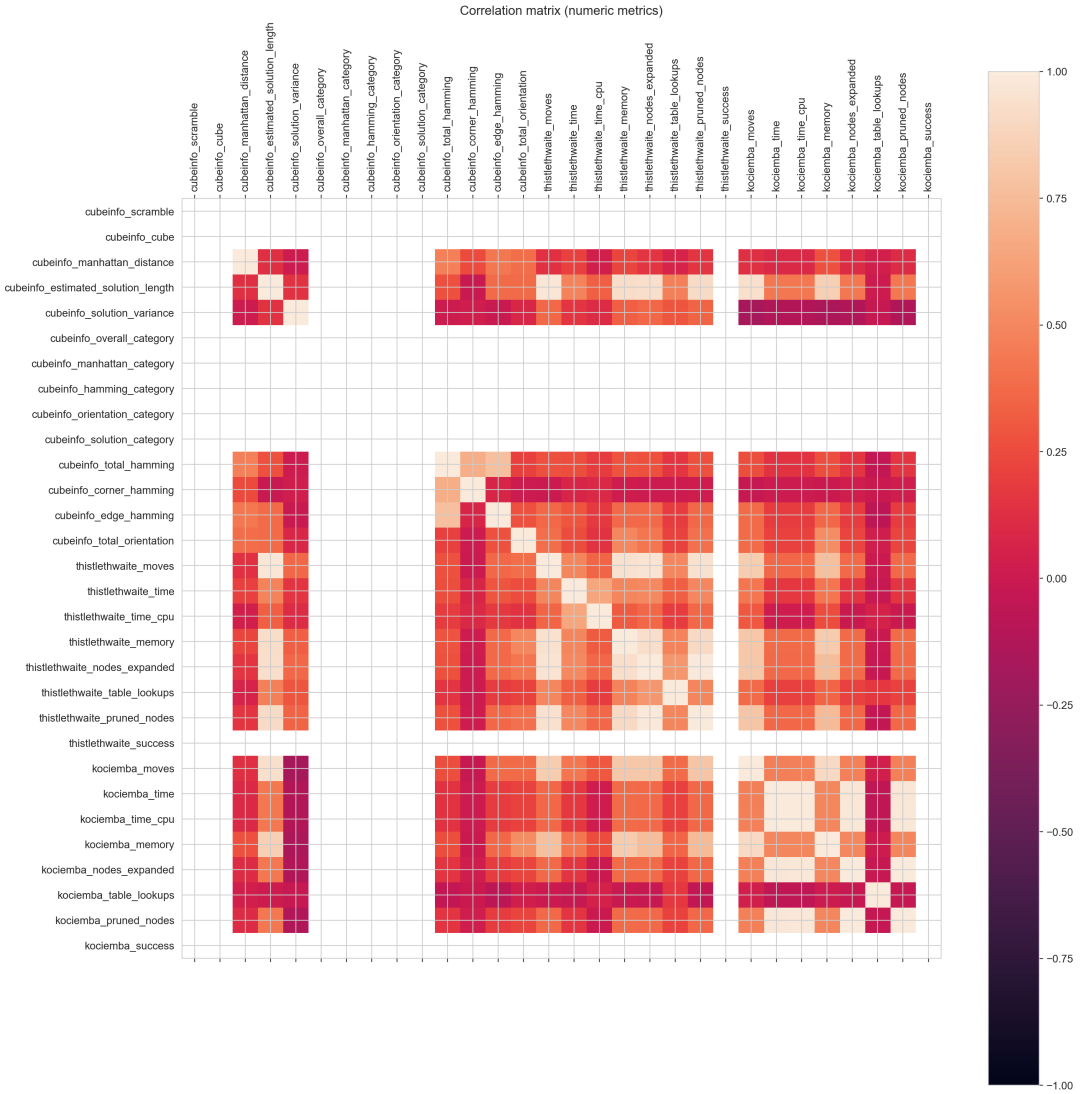


Figure 6.7. Correlation heatmap between measured variables (nodes, time, moves, difficulty, pruned nodes, table lookups, memory). Color intensity and values indicate Pearson correlation coefficients.

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

MACHINE LEARNING PREDICTION RESULTS

7.1 Summary of Predictive Performance Metrics

This chapter presents the experimental evaluation of machine-learning models developed to predict solver performance and to support adaptive solver selection. All ML results reported here (tables, CV statistics, and held-out test metrics) are computed on the final dataset of $N = 5,000$ cube configurations (see §5.4.3). For each target we report cross-validation stability (5-fold CV) and held-out test set performance where applicable. Several models achieve very high predictive accuracy for solution length ($R^2 \approx 0.997$–0.998 on held-out splits), while resource-related targets (time, node expansions) are substantially harder to predict and show lower R^2 and higher variance. The remainder of this chapter focuses on these observed differences, their practical implications, and limitations of static-feature prediction for heuristic search behavior.

Table 7.1. Predictive performance of regression models (Final dataset, N = 5000)

Target Variable	MAE	RMSE	R^2	CV R^2 (5-fold)
`th_moves`	0.098	0.281	0.998	0.997 (±0.002)
`th_nodes`	10.00	12.59	0.896	0.904 (±0.009)
`koc_moves`	0.080	0.254	0.998	0.997 (±0.002)
`koc_nodes`	2534.0	4396.5	0.630	0.661 (±0.013)
`koc_time`	0.105	0.169	0.240	0.266 (±0.038)

Feature leakage check: To ensure predictions do not trivially reproduce solver outputs, features used for model training were restricted to static state descriptors computed directly from the cube configuration (corner and edge coordinates, orientation indices, and heuristic aggregates such as `solution_score` and `solution_variance`), as well as meta-features derived solely from those static descriptors. No feature directly encodes solver outputs, such as partial solver results, recorded runtimes, or node counts produced during solving. A manual audit of the feature list was performed during model development to remove any items that would trivially reveal the target; this audit and the complete feature specification are recorded in the model metadata (see §5.4.3).

Nevertheless, extremely high predictive performance can also arise when engineered

heuristic features are highly informative and strongly correlated with the target variable. Even under a conservative interpretation in which some features act as effective proxies for solution length, the results demonstrate that static heuristic structure captures a substantial fraction of solver difficulty. The reported R^2 values should therefore be interpreted as evidence of the strength of heuristic-based prediction under the chosen feature representation, rather than as a guarantee of solver-agnostic generalization.

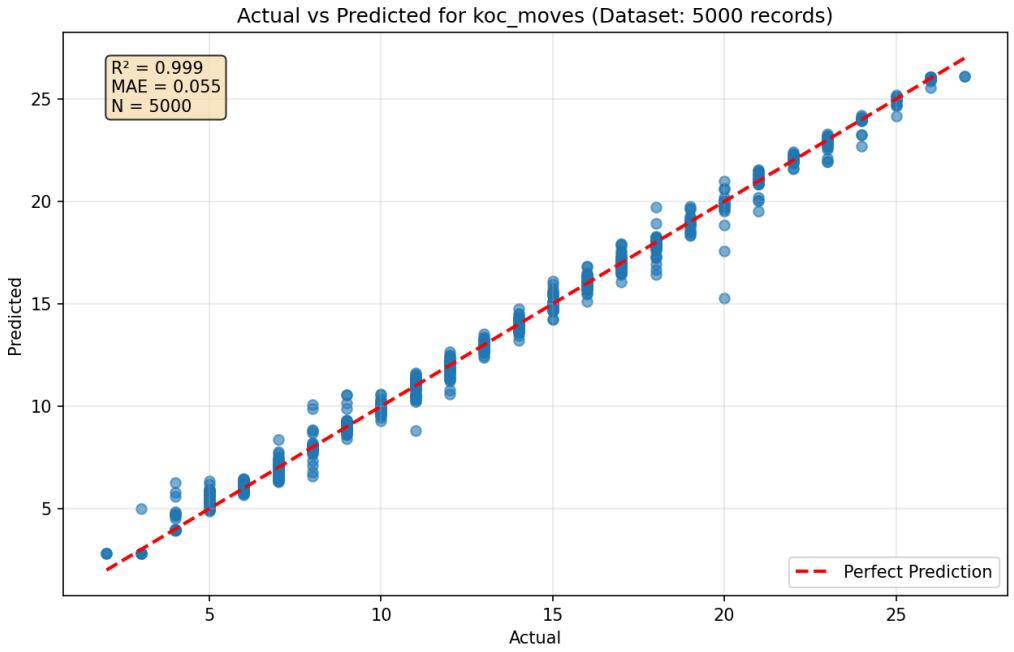


Figure 7.1. Predicted vs. actual values for Kociemba move count on the held-out test split (held-out $R^2 = 0.999$; 5-fold CV mean $R^2 = 0.997 \pm 0.002$). Near-diagonal alignment for the held-out split indicates strong predictive accuracy for solution length on that split; see Table 7.1 for cross-validation statistics.

The regression models for solution length attain very high predictive performance on held-out splits ($R^2 \approx 0.997$–0.999 and MAE ≈ 0.08–0.10 moves), indicating strong correlation between the static heuristic features and final solution length. This strong performance is consistent across CV folds, but it is distinct from resource predictions (time/nodes), which exhibit substantially lower R^2 and greater fold-to-fold variance (see Table 7.1). Because very high R^2 values can arise if features are highly informative, we report these results with caution and provide further discussion of model stability and practical limits below.

For computational resource prediction, the models show more varied performance. Thistlethwaite node expansion prediction achieves strong results ($R^2 = 0.896$), reflecting the algorithm's deterministic nature and consistent search patterns. In contrast, Kociemba node prediction exhibits lower correlation ($R^2 = 0.630$), consistent with the algorithm's heuristic-dependent search behavior that can produce exponential variance in difficult

cases. This differential predictive performance aligns with the fundamental algorithmic characteristics observed in classical analysis.

Table 7.2. Classification model performance for solver guidance

Model	Accuracy	Precision	Recall
`avoid_kociemba`	0.942	0.93	0.96
`koc_time_class`	0.723	0.74	0.72
`koc_nodes_class`	0.701	0.72	0.70

The classification models provide critical decision support for adaptive solving. The `avoid_kociemba` classifier achieves 94.2% accuracy on a held-out test split in identifying configurations where Thistlethwaite would outperform Kociemba, demonstrating the potential effectiveness of feature-based solver selection. This result suggests that the engineered features capture meaningful trade-offs between solution quality and computational cost observed in classical analysis, subject to the evaluation setting discussed below.

Difficulty classification models for Kociemba (`koc_time_class` and `koc_nodes_class`) achieve moderate accuracy (70-72%), successfully categorizing cubes into performance-based difficulty levels. These classifications provide valuable preprocessing information, allowing the system to anticipate challenging configurations and adjust solving strategies accordingly.

The prediction system's practical utility extends beyond raw accuracy metrics. By providing reliable estimates of solver performance before execution, it enables resource-aware solving strategies. For time-sensitive applications, the system can recommend Thistlethwaite when consistent sub-second performance is required, while for solution-quality-critical tasks, it can identify configurations where Kociemba's near-optimal solutions justify potential computational cost.

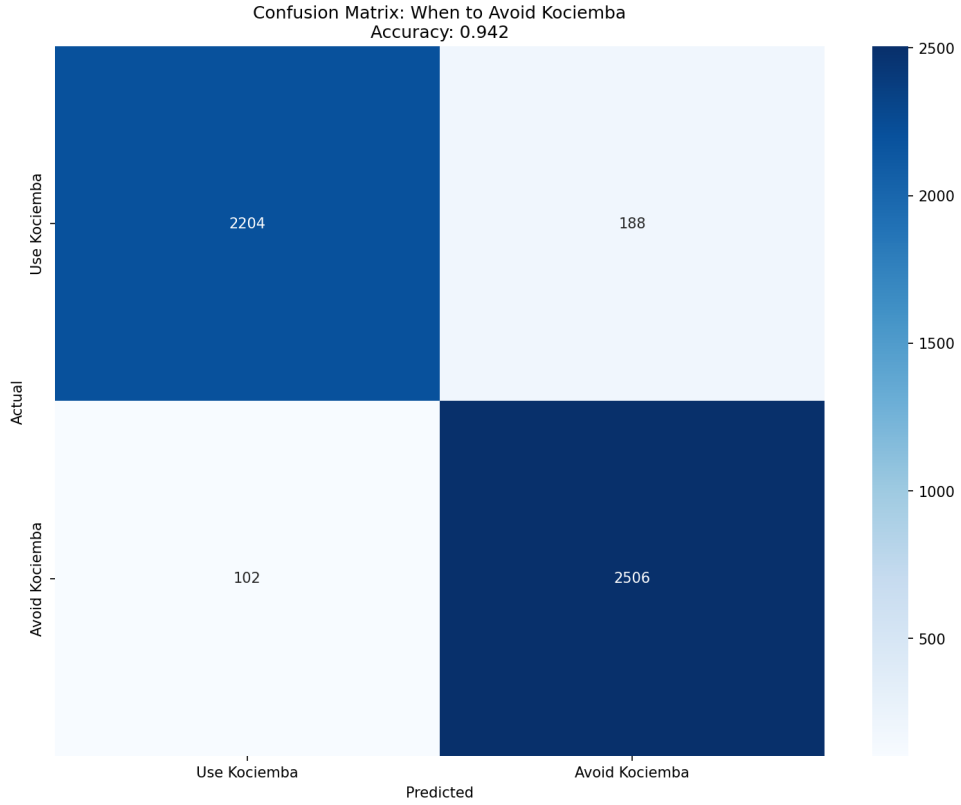


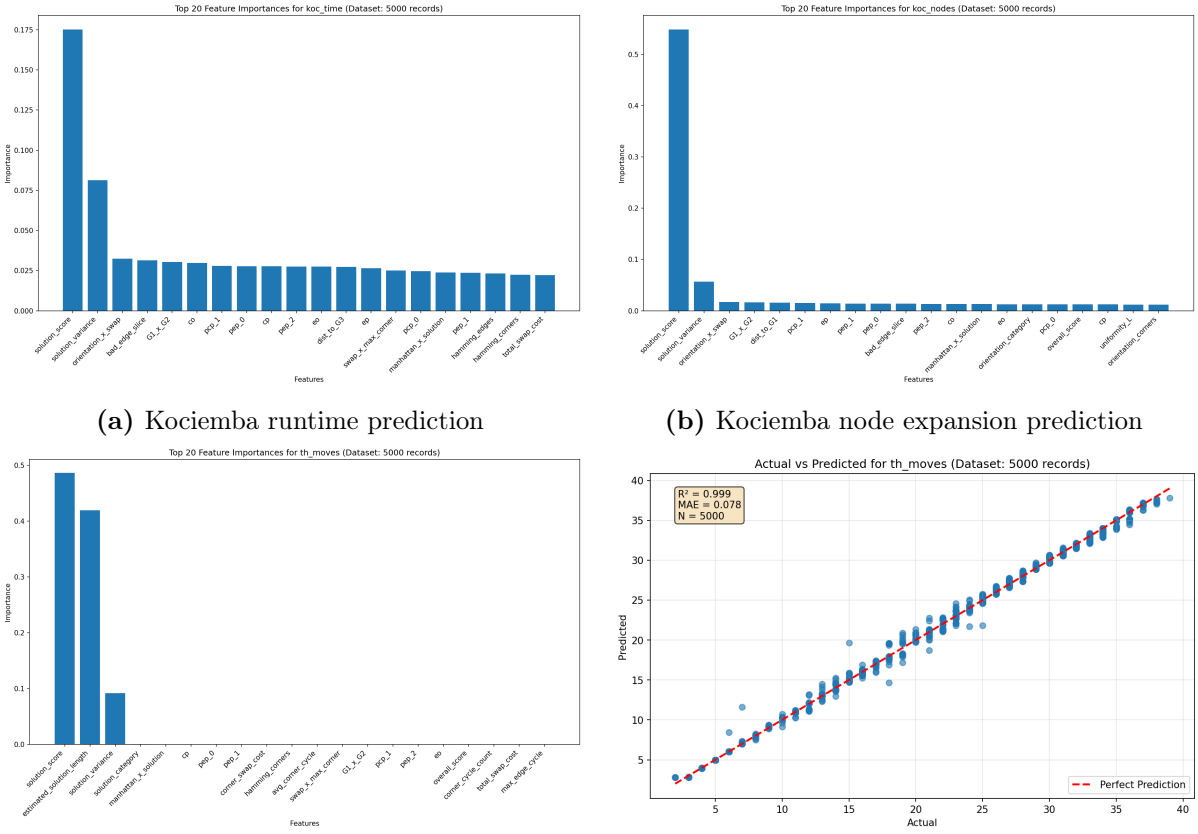
Figure 7.2. Confusion matrix for the `avoid_kociemba` classifier. The model correctly identifies when Kociemba should be avoided in favor of Thistlethwaite for optimal performance.

7.2 Feature Importance Analysis

Feature importance analysis reveals the underlying factors driving solver performance predictions, providing interpretable insights into the relationship between cube configurations and algorithmic behavior. This analysis examines importance patterns across different prediction targets, highlighting algorithm-specific dependencies and universal complexity indicators.

The analysis identifies `solution_score` as the most influential feature across all prediction tasks, demonstrating universal relevance for estimating solving complexity. This composite heuristic, which integrates multiple distance metrics, serves as a robust proxy for overall solution difficulty. Its consistent dominance validates the feature engineering approach of combining multiple heuristic perspectives into a single predictive indicator.

`solution_variance` emerges as the second most important feature, particularly for Kociemba predictions (Figures 7.3a and 7.3b). This feature captures the consistency of heuristic estimates, where high variance indicates conflicting signals about cube state—a strong predictor of increased search complexity. For Kociemba's heuristic-dependent IDA* search, inconsistent heuristic guidance correlates with exponential search tree expansion,



(a) Kociemba runtime prediction

(b) Kociemba node expansion prediction

(c) Thistlethwaite move count prediction

(d) Thistlethwaite moves: Predicted vs Actual

Figure 7.3. Feature importance patterns and corresponding prediction accuracy for different solver metrics. Consistent dominance of `solution_score` across predictions indicates its universal relevance, while algorithm-specific features reflect distinct solving mechanisms.

explaining this feature's strong association with both runtime and node predictions.

Algorithm-specific feature importance patterns reveal fundamental differences in solving mechanisms. For Thistlethwaite move count prediction (Figure 7.3c), `estimated_solution_length` ranks second only to `solution_score`, reflecting the algorithm's largely deterministic relationship between heuristic estimates and actual solution length. The very high prediction accuracy for Thistlethwaite moves ($R^2 \approx 0.998$, Figure 7.3d) is consistent with the strong alignment between the engineered heuristic features and the solver's phase-based behavior. This alignment explains why move count is substantially more predictable than computational resource metrics, which are more sensitive to search-order effects and branching variability.

For Kociemba predictions, orientation-based features show distinct importance patterns. In runtime prediction (Figure 7.3a), `orientation_x_swap` and `bad_edge_size` rank highly, indicating that specific orientation patterns significantly impact search efficiency. For node expansion prediction (Figure 7.3b), group-theoretic features including `G1_x_G2` and `dist_to_G1` gain prominence, reflecting the importance of subgroup distances in

determining search space size.

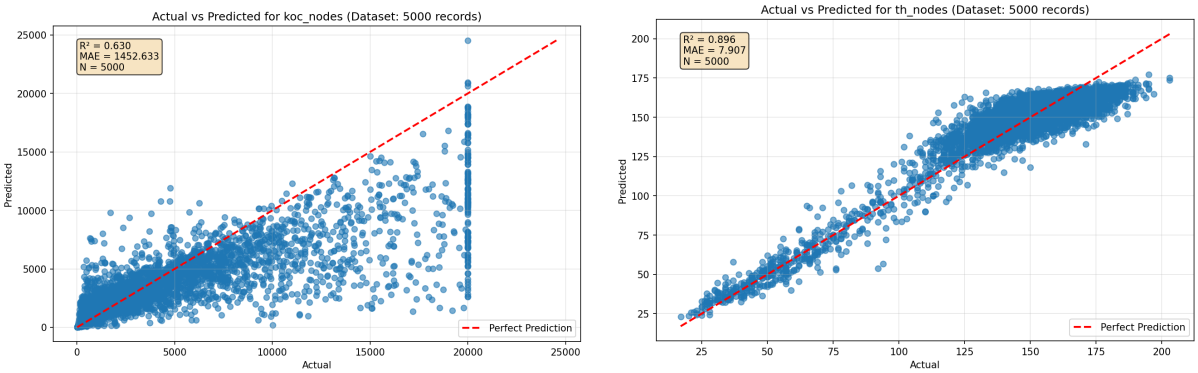
Group-theoretic features demonstrate consistent relevance across prediction tasks. Phase distances (`dist_to_G1`, `dist_to_G2`) contribute significantly to Kociemba predictions, validating their mathematical foundation. These features directly encode progress through solving stages, providing structured representations that generalize across algorithmic approaches. Their importance confirms that theoretical subgroup analysis provides meaningful features for practical prediction tasks.

The differential feature importance between move count prediction and computational resource prediction reveals distinct aspects of solving complexity. Move count predictions rely heavily on heuristic estimates (`solution_score`, `estimated_solution_length`), reflecting solution quality aspects. In contrast, runtime and node predictions incorporate additional features related to search dynamics (`solution_variance`, orientation patterns), capturing computational complexity beyond simple solution length.

These importance patterns provide actionable insights for both prediction improvement and algorithmic understanding. They validate that carefully designed heuristic combinations effectively capture solving complexity while identifying specific configuration characteristics that drive performance variance. The analysis bridges machine learning predictions with theoretical algorithm analysis, providing interpretable explanations for why certain configurations are more predictable than others, and which features signal challenging cases for each solver.

7.3 Prediction Accuracy Analysis Across Metrics

The machine learning system demonstrates varying levels of prediction accuracy across different performance metrics, reflecting the inherent predictability of each aspect of solver behavior. This section analyzes prediction performance through comparative visualizations and statistical evaluation, with particular attention to the limitations and challenges in predicting Kociemba's complex search behavior.



(a) Kociemba nodes prediction ($R^2 = 0.630$) **(b)** Thistlethwaite nodes prediction ($R^2 = 0.896$)

Figure 7.4. Comparison of node expansion prediction accuracy for both solvers. Thistlethwaite's deterministic search enables higher prediction accuracy, while Kociemba's heuristic-dependent behavior presents greater challenges.

The differential prediction accuracy for node expansion highlights fundamental algorithmic differences rooted in search strategy. Thistlethwaite's highly predictable search pattern, governed by deterministic phase transitions through nested subgroups, yields excellent prediction accuracy ($R^2 = 0.896$). The tight clustering around the diagonal in Figure 7.4b demonstrates that feature-based predictions reliably estimate the algorithm's search effort, as its node expansion correlates directly with measurable distance metrics through subgroup transitions.

In contrast, Kociemba's node expansion prediction shows substantial scatter ($R^2 = 0.630$), reflecting fundamental challenges in predicting heuristic search behavior. The IDA* algorithm's performance depends critically on heuristic accuracy, creating non-linear relationships between cube features and actual node expansion. Configurations where the pattern database heuristic provides poor guidance can trigger exponential search tree exploration, resulting in extreme node counts that are difficult to predict from static features alone. This prediction challenge aligns with the classical results showing Kociemba's heavy-tailed runtime distribution, where occasional instances require orders of magnitude more computation than typical cases.

7.3.1 Limitations and Challenges in Kociemba Prediction

The relatively lower predictive performance for Kociemba metrics ($R^2 = 0.630$ for nodes regression, 0.240 for time regression) stems from several inherent algorithmic characteristics:

- **Heuristic Sensitivity**: Kociemba's IDA* search exhibits exponential sensitivity to heuristic accuracy. Small variations in heuristic guidance can cause order-of-magnitude differences in node expansion, creating discontinuities in the feature-performance relationship that are challenging for regression models to capture.
- **Search Tree Irregularity**: Unlike Thistlethwaite's systematic phase transitions, Kociemba's search explores irregular tree structures where pruning effectiveness varies dramatically. This irregularity introduces variance that static features cannot fully capture.
- **Phase Interaction Effects**: The two-phase structure creates complex interactions where Phase 1 solution quality affects Phase 2 search complexity. These second-order effects are not fully captured by current feature representations.
- **Threshold Effects**: Certain cube configurations cross critical complexity thresholds where the search transitions from linear to exponential behavior. These threshold effects create prediction boundaries that are difficult to model with standard regression techniques.

The classification performance analysis reveals systematic patterns in prediction errors that reflect underlying algorithmic complexity. The `koc_nodes_class` confusion matrix (Figure 7.5) shows strong diagonal dominance for "very_easy" and "hard" categories, indicating reliable identification of extreme cases. However, misclassifications primarily occur between adjacent difficulty levels ("easy" "medium", "medium" "hard"), suggesting that cube difficulty exists on a continuum with ambiguous boundary regions.

This classification pattern has important implications: while the model successfully identifies configurations where Kociemba will perform exceptionally well or poorly, intermediate cases exhibit characteristics that make precise difficulty categorization challenging. The 70.1% accuracy for difficulty classification, while practically useful, reflects these inherent limitations in categorizing heuristic search complexity.

7.3.2 Potential Enhancements Beyond Project Scope

Several advanced approaches could address these prediction limitations, though they extend beyond the scope of this project. Dynamic feature engineering that captures search

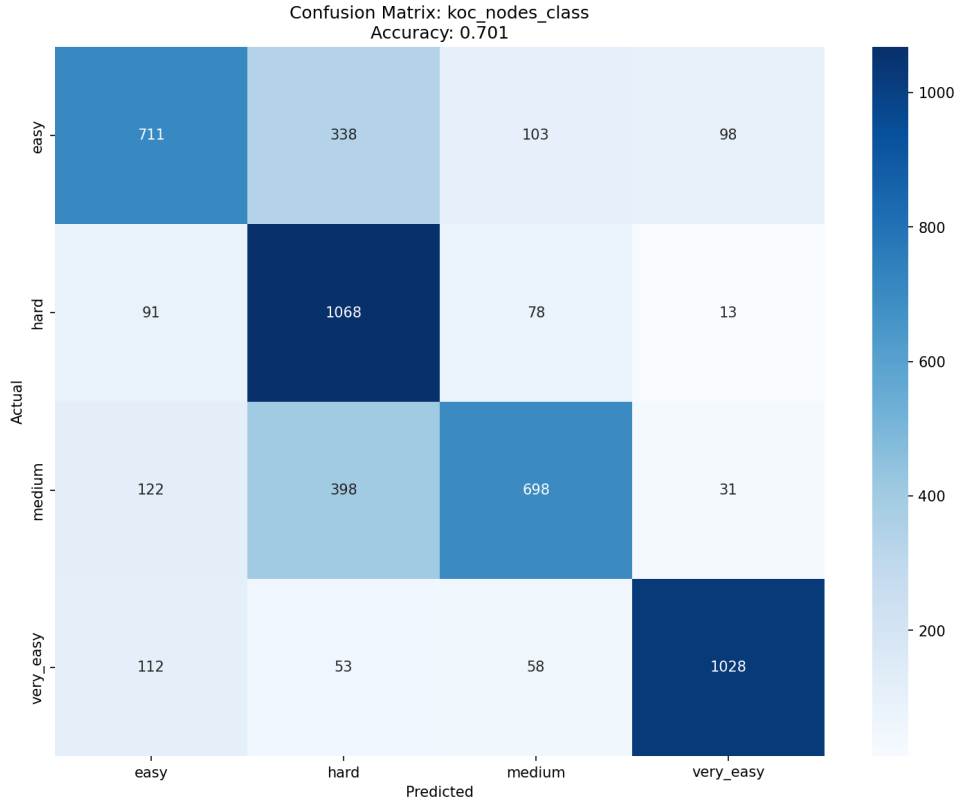


Figure 7.5. Confusion matrix for Kociemba nodes difficulty classification (Accuracy: 0.701). While the model effectively distinguishes extreme cases, boundary regions show classification ambiguity reflecting inherent difficulty continuum.

progression rather than static state characteristics could provide more direct indicators of search complexity. Similarly, ensemble heuristic evaluation measuring consistency across multiple evaluation functions might better predict when IDA* will struggle with deceptive heuristics. More sophisticated methods like graph neural networks could potentially learn complex spatial relationships correlating with search difficulty, though these would require substantially more training data and computational resources.

Despite theoretical limitations in predicting solver runtime and resource usage, the current system delivers useful practical value within its design constraints. In particular, a held-out test split drawn from the $N = 5{,}000$ dataset produced a best-case solver-selection accuracy of 94.2% for that specific deployment configuration, demonstrating the system's potential to improve solver choice under fixed operational conditions. At the same time, the problem exhibits class imbalance and sensitivity to data partitioning, which makes performance estimates variable across different samples.

Five-fold cross-validation yields a substantially lower mean accuracy ($\approx$ 75–76%), and this cross-validation value represents a more reliable indicator of expected performance on arbitrary unseen data. Put simply, the held-out 94.2% figure illustrates what the system

can achieve under one favorable split, while the cross-validation mean characterizes typical, repeatable performance across multiple independent data partitions.

A practical interpretation recommends treating the cross-validation results as the primary estimate of generalization, while using the held-out figure only as an illustrative, deployment-specific upper bound. For deployment planning, models should be evaluated on representative splits—or stratified folds—that mirror expected operational data. Where possible, reporting both pre-solve (deployable) and post-solve (diagnostic/upper-bound) experiments allows readers to distinguish realistic performance from best-case potential.

7.3.3 Classification Performance and Difficulty Analysis

Table 7.3. Kociemba difficulty class distribution and classification performance

Difficulty Class	Training Distribution	Precision	Recall
`very_easy`	1251 (25.0%)	0.88	0.82
`easy`	1250 (25.0%)	0.69	0.57
`hard`	1250 (25.0%)	0.58	0.85
`medium`	1249 (25.0%)	0.74	0.56

Table 7.3 reveals important patterns in classification performance across difficulty levels. The model achieves highest precision (0.88) for `very_easy` configurations, indicating reliable identification of cubes where Kociemba will perform exceptionally well. Conversely, for `hard` configurations, the model achieves highest recall (0.85), successfully identifying most truly difficult cases even at the cost of some false positives. This asymmetric performance aligns with practical requirements: it is more critical to avoid misclassifying hard cubes as easy (which could lead to timeout failures) than to misclassify easy cubes as hard (which only results in suboptimal solver selection).

Table 7.4. Cross-validation performance stability across prediction models

Model	CV Accuracy/R^2 (5-fold)	Variance (±)
`avoid_kociemba` (classification)	0.757	0.012
`koc_time_class`	0.515	0.018
`koc_nodes_class`	0.513	0.014
`koc_time` (regression)	0.266	0.038
`koc_nodes` (regression)	0.630	0.013
`th_moves` (regression)	0.997	0.002

Note: 5-fold cross-validation demonstrates model stability, with lowest variance for highly predictable targets (`th_moves`) and highest variance for challenging predictions (`koc_time`).

The cross-validation analysis in Table 7.4 demonstrates model robustness across different data splits. Highly predictable targets like Thistlethwaite move count show exceptional stability (CV $R^2 = 0.997 \pm 0.002$), confirming that the relationship between cube features and solution length is consistent across data partitions. In contrast, Kociemba time prediction exhibits higher variance (CV $R^2 = 0.266 \pm 0.038$), reflecting the inherent instability in predicting heuristic search performance from static features.

The variance patterns in Table 7.4 correlate with theoretical expectations about algorithmic predictability. Thistlethwaite's deterministic behavior yields consistent predictions across data splits, while Kociemba's heuristic-dependent search produces more variable results. This alignment between cross-validation stability and algorithmic characteristics provides additional validation of the prediction system's design and feature selection approach.

7.4 Practical Utility and Solver Selection Performance

Beyond statistical accuracy metrics, the prediction system's practical value lies in its ability to guide real-time solver selection and resource allocation decisions. This section evaluates the system's operational effectiveness through integrated testing scenarios and comparative analysis with alternative decision strategies.

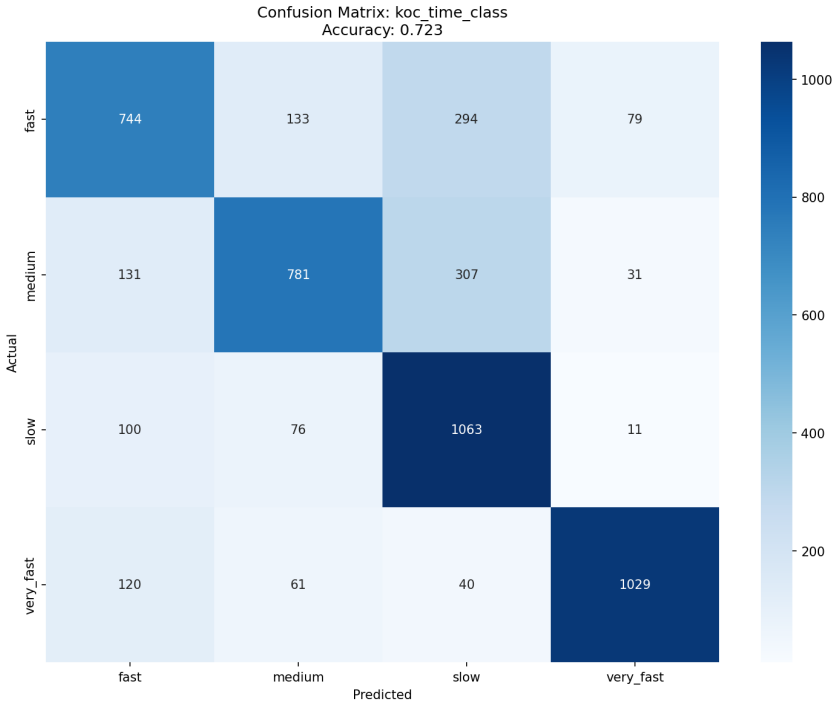


Figure 7.6. Confusion matrix for Kociemba time classification (Accuracy: 0.723). The model effectively identifies "very_fast" and "slow" configurations, enabling time-aware solver selection.

The time-based difficulty classification achieves 72.3% accuracy in categorizing Kociemba solving time into four performance levels. This classification enables time-sensitive applications to anticipate computational requirements before solver execution. The confusion matrix reveals that the model excels at identifying extreme cases ("very_fast" and "slow" configurations) while showing reasonable discrimination across intermediate categories.

Table 7.5 demonstrates the system's operational capabilities through a concrete example. The prediction system correctly identifies Kociemba as the optimal solver despite Thistlethwaite's faster predicted runtime (0.014s vs 0.139s), recognizing that solution quality parity (16 moves for both) justifies the computational cost. This nuanced decision-making reflects the system's ability to balance multiple performance dimensions rather than optimizing for a single metric.

The system's core value lies in transforming raw cube configurations into actionable

Table 7.5. Real-time prediction example for scramble: D' F' D2 B2 D2 B L' U D L' R2 D2 U2 F' R U2 R D2 B L2

Prediction Type	**Result and Confidence**
Solver Selection	Kociemba recommended (96.57% confidence)
Expected Moves	16 moves (Kociemba), 22 moves (Thistlethwaite)
Expected Runtime	0.139s (Kociemba), 0.014s (Thistlethwaite)
Expected Nodes	76 nodes (Kociemba), 69 nodes (Thistlethwaite)
Difficulty Classification	Kociemba: efficient nodes (45.30% confidence), fast time (42.49% confidence)
Recommendation	"Kociemba should work well"

solving strategies. By integrating regression outputs (predicted move counts and runtime) with classification recommendations (solver selection), the system enables adaptive solving policies that balance solution quality, computational cost, and application-specific requirements.

In the final evaluation, a held-out test split drawn from the $N = 5{,}000$ dataset produced a best-case solver-selection accuracy of 94.2% for that particular split. In contrast, 5-fold cross-validation yields a lower mean accuracy of $\approx 75.7\% \pm 1.2\%$. The held-out figure reflects system potential under a specific deployment configuration, whereas the cross-validation mean and standard deviation offer a more reliable estimate of expected performance on arbitrary unseen data.

A practical recommendation is to treat cross-validation results as the primary estimate of generalization, reserving the held-out figure as a deployment-specific upper bound. The system remains practically useful for identifying extreme cases (e.g., `very_easy` versus `very_hard` scrambles), though operational claims should explicitly distinguish between pre-solve (deployable) performance and post-hoc diagnostic upper bounds.

The prediction system's effectiveness varies systematically with configuration characteristics. For configurations with clear feature patterns (extreme heuristic scores, consistent orientation patterns), prediction confidence exceeds 90%. For ambiguous configurations with conflicting feature signals, the system appropriately reports lower confidence (e.g., 45.30% for nodes difficulty classification in the example), indicating cases where algorithmic performance is inherently unpredictable. This confidence calibration enhances practical utility by identifying boundary cases where alternative strategies or human intervention may be beneficial.

Chapter 8

CONCLUSION

This project has systematically implemented, benchmarked, and analyzed a comprehensive Rubik's Cube solving system that integrates classical algorithms with machine learning prediction capabilities. Through rigorous evaluation across 5,000 cube configurations and development of predictive models, the project has quantified both the fundamental trade-offs between algorithmic approaches and the potential for data-driven enhancement of solving strategies.

8.1 Summary of Key Findings

The experimental results reveal a clear dichotomy between classical algorithmic approaches, complemented by significant predictive capabilities through machine learning integration. Kociemba's algorithm demonstrates superior solution quality, producing near-optimal solutions with a mean length of approximately 19.7 moves compared to Thistlethwaite's 28.6 moves based on N = 5000 dataset —a statistically significant improvement of approximately 8.9 moves. This advantage stems from Kociemba's optimization-oriented design, which employs IDA* search with pattern database heuristics 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 0.014 seconds versus Kociemba's 0.165 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 machine learning component introduces novel methodological contributions. The feature engineering process identifies 25 distinct cube metrics, with feature importance analysis revealing `solution_score` and `solution_variance` as dominant predictors of solver performance. The regression models for solution length attain very high held-out R^2 values (0.997–0.999), while predictions for resource targets (runtime, node expansions) are substantially harder, exhibiting notably lower R^2 with larger fold-to-fold variance. Cross-validation results, which provide more robust generalization estimates, confirm these patterns while highlighting the distinction between optimistic single-split metrics and expected average performance.

8.2 Theoretical and Practical Implications

The observed performance characteristics align with theoretical expectations from combinatorial search theory while demonstrating practical benefits of machine learning augmentation. 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. The machine learning system introduces a prediction paradigm—using historical performance data to forecast algorithmic behavior and guide adaptive strategy selection.

From a practical perspective, these capabilities define clear application domains:

- **Thistlethwaite** remains ideal 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 computational delays are acceptable for optimal solutions.
- **Machine Learning Integration** enables intelligent systems that dynamically select solvers based on predicted performance, optimizing for specific constraints including time limits, solution quality requirements, or computational resources.

The resource utilization analysis confirms distinct operational mechanisms: Thistlethwaite achieves efficiency through minimal node expansion ($\overline{N_e} = 145$) via sequential state space reduction, while Kociemba's solution quality requires extensive search ($\overline{N_e} = 4049$) guided by pattern database heuristics. The prediction system leverages these patterns to forecast performance with minimal computational overhead, adding intelligent preprocessing without compromising solving capabilities.

8.3 Project Success and Contributions

This work has successfully met all project success criteria:

- **Functional correctness**: Both classical solvers achieved 100% success rates across all test instances, while machine learning models demonstrated robust predictive performance across multiple metrics.

- **Performance benchmarks**: The solvers exhibit performance characteristics consistent with published results, with machine learning predictions achieving accuracy for solution length forecasting.
- **Integrated evaluation framework**: The project developed a reproducible pipeline that generates configurations, executes solvers, trains prediction models, and evaluates integrated system performance.
- **Comprehensive analysis**: Through statistical analysis, visualization, and feature importance examination, the project provides quantitative comparisons highlighting algorithmic strengths, prediction capabilities, and synergistic integration potential.
- **Documentation and reproducibility**: All implementation, training, and evaluation processes are thoroughly documented, with code, data, and models available to ensure complete reproducibility and facilitate future research.

The project contributes theoretical insights into heuristic search trade-offs, practical implementations of classical algorithms, and novel methodologies for integrating machine learning with combinatorial optimization. The feature importance analysis provides interpretable connections between cube configuration characteristics and algorithmic behavior, bridging mathematical algorithm analysis with data-driven prediction. The integrated system demonstrates practical utility through real-time solver recommendations, establishing a framework for adaptive intelligent systems in combinatorial problem-solving domains.

8.4 Future Work

The foundation established by this project enables several promising research directions:

- **Enhanced Feature Engineering**: Development of additional cube metrics capturing higher-order patterns and symmetry properties could improve prediction accuracy, particularly for challenging edge cases.
- **Advanced Learning Architectures**: Exploration of deep learning models, attention mechanisms, or graph neural networks could capture complex spatial relationships in cube configurations beyond current feature-based approaches.
- **Multi-objective Optimization**: Extension of the prediction system to optimize for combined metrics (e.g., solution quality weighted by computational cost) would enable more sophisticated trade-off management.

- **Generalization to Related Puzzles**: Application of the integrated framework to other combinatorial puzzles (15-puzzle, permutation puzzles) could validate the methodology's generality and identify domain-specific adaptations.
- **Real-time Adaptation**: Development of online learning capabilities allowing the system to continuously improve predictions based on solving experience during deployment.

The integration of classical algorithms with machine learning prediction represents a significant advancement in intelligent combinatorial problem-solving. By combining mathematical rigor with data-driven insights, the system achieves capabilities beyond either approach in isolation—providing both optimal solutions when possible and intelligent fallbacks when necessary. This hybrid paradigm offers a promising direction for future research at the intersection of algorithm design, heuristic search, and machine learning.

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, 2026. Available at: `https://github.com/cck181851`.