Heuristic Function Classification using Machine Learning

Aaryan Koulavkar

April 2025

Abstract

This work investigates the classification of heuristic functions used in A* search for 8-puzzle problems using machine learning. The goal is to identify which heuristic—Manhattan, Euclidean, Hamming, or Linear Conflict—was employed in a given search trace by analyzing structural and statistical features of the search. A synthetic dataset of 10,000 diverse, solvable episodes was generated through randomized start and goal configurations, ensuring statistical parity and generalization. Features were extracted from each A* execution capturing graph topology, search dynamics, and heuristic behavior—including solution length, node expansions, branching factor, heuristic variance, and directionality of state transitions. Multiple classifiers were trained including Random Forest, MLP, XGBoost, and LightGBM, and the best-performing model was a StackingClassifier trained using filtered SHAP-important features and class-weight adjustments, achieving a validation accuracy of 88%. Label encoding and serialization pipelines ensured reproducibility and consistent evaluation. Feature importance analysis reveals clear separability between heuristics in terms of efficiency and smoothness. This research contributes to explainable heuristic modeling and opens pathways to dynamic heuristic introspection in search algorithms.

1 Synthetic Data Generation

To classify heuristics used in A* search on 8-puzzle problems, we began by generating a synthetic dataset through randomized episodes. The goal was to simulate A* search behavior for different heuristics—Euclidean, Manhattan, Hamming, and Linear Conflict—under varied start and goal configurations to create a diverse, unbiased training dataset. The complete synthetic data generation pipeline was designed to ensure statistical representativeness, minimize bias, and support generalization.

1.1 Algorithmic Steps

1. Define the solved state as `(1, 2, 3, 4, 5, 6, 7, 8, 0)`.
2. For each episode:

(a) Generate a random goal state via 15–25 random valid moves from the solved state.

(b) Generate a start state via 20–30 random moves from the goal state to ensure solvability.

(c) Randomly sample one of the heuristics from the `HEURISTICS` dictionary.

(d) Execute the A* algorithm and store the episode only if the goal is reachable.

3. Compute extended statistical and structural features using `extract_features_extended()`.

4. Convert NumPy objects to native Python types for JSON serialization.

5. Encode the heuristic name using `LabelEncoder()` and persist the mapping using `joblib.dump()`.

1.2 Noise Reduction and Generalization

To ensure high-quality data:

- Episodes where A* failed to find a solution were skipped to avoid label noise.
- Randomization of both the start and goal states ensures that heuristics are tested across wide topological diversity.
- By creating complex state transitions, we simulate real-world variations in solution depth, branching factors, and suboptimality.
- JSON serialization compatibility was handled by recursively converting NumPy datatypes to native Python types, preserving model compatibility.
- LabelEncoder ensures categorical consistency and allows meaningful decoding of predictions during evaluation.

This process produced 10,000 clean, valid, and diverse episodes across all four heuristics, ensuring statistical parity and eliminating spurious correlations. This supports generalization beyond memorized patterns and facilitates learning high-level heuristic behaviors.

2 Model Construction and Training

The model architecture includes:

- **Random Forest Classifier**: Trained using class-weight balancing and hyperparameter tuning (`GridSearchCV`).
- **MLPClassifier**: A feedforward neural network trained with adaptive early stopping.
- **XGBoost**: Gradient boosting classifier, optimized with loss and depth tuning.
- **LightGBM**: Histogram-based learner, enhanced with `class_weight='balanced'`.

- **StackingClassifier**: Combines predictions from base models using a meta-learner (Logistic Regression).

To address class imbalance, we used `compute_class_weight` from scikit-learn to assign balanced weights to underrepresented classes. A SHAP-based filter was applied to select the most important 80% percentile features. The stacking model was retrained on these reduced dimensions. Accuracy on the validation set was **88.00%**, with permutation importance confirming consistency in influential features.

3 Label Encoding and Evaluation

To maintain consistency across the pipeline:

- `LabelEncoder()` was trained on the full set of heuristic names.
- The encoder was stored with the final model using `joblib.dump()`.
- During inference or evaluation, predictions were decoded back to string labels via the encoder.
- This ensured human-readable outputs and avoided mismatches in mapping during evaluation.

4 Detailed Feature Justification

The following paragraphs provide in-depth justification for each of the 28 features used in classification. These features are extracted from the solution trace, search graph, and heuristic distribution statistics, and were selected based on their ability to model heuristic behavior under theoretical frameworks.

4.1 1. Solution Length

Indicates the cost of the optimal path C^*. A heuristic with higher admissibility (e.g., Manhattan) will track C^* more closely. Statistically, longer paths under weak heuristics introduce more noisy expansions.

4.2 2. Number of Expanded Nodes

Correlates inversely with heuristic informativeness. Probabilistically, better heuristics reduce entropy in the frontier. Hamming often has higher node expansion.

4.3 3. Maximum Search Depth

Tied to tree depth and indirectly to heuristic consistency. Euclidean typically causes deeper expansions due to smooth gradients.

4.4 4. Average Branching Factor

Higher values imply less effective pruning. Statistically, branching factor variance reveals structural gaps in the heuristic guidance.

4.5 5. Average Heuristic Value

Reflects the global optimism of the heuristic. Higher average implies conservative estimation—Linear Conflict tends to inflate cost.

4.6 6. Heuristic Variance, Max, Min

Represents dispersion. Higher variance (e.g., Hamming) implies inconsistent progress; lower variance (e.g., Manhattan) indicates more stable estimation.

4.7 7. Goal State Visit Ratio

Measures proximity to goal across the path. A heuristic like Manhattan encourages early alignment, raising this score.

4.8 8. Clustering Coefficient

From network theory, measures local density. Better heuristics cluster nodes along the optimal path; Hamming leads to sparse graphs.

4.9 9. Solution Efficiency Ratio

Defined as $\frac{C^*}{\text{Nodes Expanded}}$, this gives a probabilistic efficiency score. Higher values indicate better heuristic precision.

4.10 10. Forward vs Backward Transitions

Analyzes the net direction of state transitions. Strong heuristics tend to induce forward motion; weak ones cause looping.

4.11 11. Time Taken

Wall-clock time indicates computational efficiency. Often correlates with expansions and indirectly with heuristic quality.

4.12 12. Heuristic Delta (Smoothness)

Measures mean gradient between successive heuristic values. A consistent heuristic has lower deltas; Euclidean and Manhattan exhibit this behavior.

These features were chosen because they exhibit both statistical and algorithmic separability across heuristics. For instance, Hamming distance produces jagged heuristic landscapes and erratic expansions, resulting in high variance, high branching, and poor solution efficiency. In contrast, Manhattan and Euclidean heuristics exhibit smoother heuristic transitions, lower deltas, fewer node expansions, and stronger forward directionality. These measurable differences, grounded in theory, were what the model leveraged to discriminate effectively between heuristics.

5 Conclusion

This study successfully demonstrates the viability of using machine learning to classify heuristic functions in A* search applied to the 8-puzzle problem. By designing a well-structured synthetic data pipeline, incorporating extensive feature engineering, and leveraging ensemble learning with SHAP-based pruning and class rebalancing, we achieved a validation accuracy of 88.00