Suboptimal Multi-Heuristic Approaches for Solving the Rubik's Cube Incorporating Deep Learning and Group Theory

A Thesis
Presented to
The Established Interdisciplinary Committee for
Mathematics and Computer Science
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Yancheng Liu

May 2024

Approved for the Committee
(Mathematics and Computer Science)

Greg Anderson	David Meyer

Acknowledgements

I would like to extend my deepest gratitude to my parents, whose unwavering support and wise counsel have been instrumental in my achievements. Their respect for my choices, including my decision to attend Reed, has profoundly shaped my journey.

I am immensely thankful to my advisors, Greg Anderson and David Meyer, for their invaluable guidance and dedication in enhancing this thesis. This extensive work spans multiple disciplines, and their willingness to meticulously review iterations and meet during their free time has provided a solid foundation for its success.

Special thanks are also due to my research mentor, Anna Ritz. Her mentorship has transcended academic boundaries, offering insights into both my professional and personal life. She was the first to support my thesis on the Rubik's cube, providing positive feedback that paved the way for this project. Without her, this thesis would not have been possible. I am particularly grateful for the access to the RTX 3090 in the Bio Lab, which significantly expedited my research, a task that would have been far more time-consuming with my RTX 2060.

Lastly, I want to acknowledge my friends who have stood by me during the long hours at the thesis table. Your companionship and support have made this challenging process a joyous journey. I will cherish every memory of Korean BBQ, Din Tai Fung, and The One BBQ meals we shared, and I will forever remember the laughter and experiences we enjoyed together.

List of Notation

$\mathbb{Z}_3^8$	The group of 8-tuples over $\mathbb{Z}_3$
$[x]_n$	The value of x mod n
G	The Rubik's Cube Group
S	The set of all possible cube states
VS	The set of all valid cube state
Ω	The set of generating moves for G
Ψ	The set of all allowed moves
∇	The Laplacian operator or gradient operator
$R(z)$	ReLU activation function acting on z
$A(\boldsymbol{X})$	Applies the ReLU activation function to matrix $\boldsymbol{X}$
$P(s', s \mid a)$	The probability of transitioning from state s to s' via action a
$R(s, a, s')$	The reward received after transitioning from state s to s' via action a
π^*	The optimal policy for a Markov Decision Process (MDP)
V_π^*	The value function with respect to π^*
$\hat{V}(\cdot, \theta)$	The optimal state-value function approximation with the parameters θ
$\text{sgn}(\sigma)$	The sign of the permutation σ
$\ker(f)$	The kernel of the function f
$\text{im}(f)$	The image of the function f
$\text{null}(f)$	The nullity of the function f
$\dim(f)$	The dimension of the function f
$\text{domain}(f)$	The domain of the function f
$O(f)$	The big-O notation describing the asymptotic behavior of f

List of Abbreviations

IDA*	Iterative Deepening A* Search
MCTS	Monte Carlo Tree Search
GBFS	Greedy Best-First-Search
MSE	Mean Squared Error
CEE	Cross-Entropy Error
DRL	Deep Reinforcement Learning
DSL	Deep Supervised Learning
MBWA*	Multi-heuristic Batched Weighted A* Search
MBS	Multi-heuristic Beam Search
MAWA*	Multi-heuristic Anytime (Batched) Weighted A* Search

Table of Contents

Abstract

This thesis explores efficient algorithms for solving the Rubik's Cube by balancing solution quality with computational constraints. The study integrates group theory and deep learning to enhance heuristic algorithms, aiming for suboptimal solutions under specific conditions: rapid solving within 26 moves in quarter-turn metric and finding the shortest solution within a 60-second computational limit. Additionally, this research introduces a robust algorithm designed to validate the solvability of Rubik's Cube states, employing group theory to analytically dissect and understand the cube's theoretical structures.

The thesis reviews heuristic search methods and proposes a novel model combining deep reinforcement learning with supervised learning approaches. This model serves as an advanced heuristic function to direct informed search algorithms toward more efficient solutions faster. Using the improved heuristic function and group theory along with other improvements to A* search and beam search, this thesis implements Multi-Heuristic Batched Weighted A* Search (MBWA*), Multi-Heuristic Beam Search (MBS), and Anytime Variation of MBWA* (MAWA*) for different objectives. These methods are tested through comparative studies to assess their effectiveness in solving the cube under the stated constraints.

This work contributes to computational theory and algorithm design by demonstrating the potential of merging theoretical mathematics with practical computational techniques to address complex problems. The findings could lead to more sophisticated algorithms that might be applicable in other complex systems beyond Rubik's Cube, pushing the boundaries of what can be achieved within stringent computational and time constraints.

Chapter 1

Introduction

1.1 Motivation

My journey with problem-solving began in junior high when I mastered the Rubik's Cube, achieving a 15-second solve time within a year. This early fascination blossomed into a deeper exploration of its mathematical intricacies, culminating in a high school graduation essay based on group theory. After graduation from high school, my satisfaction stemmed not from the speed of solving the cube, but from how I could solve it in the fewest steps in the most optimized manner. This challenge is the real essence of the Rubik's Cube to me, transcending mere memorization and execution of algorithms. This led me to computer science, where I initially planned to create a vast graph to record all possible cube states and solve it in minimal moves using Breath First Search. However, with 43 quintillion states, this would take an unfeasible three million years on a regular computer. This realization didn't deter me; instead, it sharpened my interest and ambition in seeking out and devising an efficient algorithm capable of solving the cube within a practical timeframe or solution length, especially within the constraints of limited resources and computing power.

1.2 Background Information

The Rubik's Cube, invented in 1974 by Hungarian architect Ernö Rubik, is a 3D combination puzzle that has become one of the world's best-selling toys. Originally designed to help students grasp three-dimensional geometric concepts, the Cube consists of a rotating mechanism that allows each face to turn independently. Marketed globally as the "Rubik's Cube" in 1980, it not only became a challenging puzzle but also a cultural icon. In the world of cubing, two metrics stand paramount: solving time and solution length, each celebrated through official World Cube Association (WCA) competitions. In the aspect of speed-solving, Max Park achieved a 3.17-second solve in 2023 which set the current world record and Yiheng Wang set the 4.48-second average world record in the same year. The fewest move count challenge demands solving the cube in minimal steps within 60 minutes. The current record stands at an average of 20 steps across three solves. [Wor24]

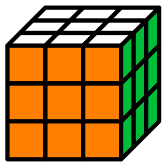


Figure 1.1: Solved State of the Rubik's Cube

The Rubik's Cube is more than just a mere puzzle or toy that poses significant challenges in computational theory, applied mathematics, and artificial intelligence. Its complexity has prompted extensive theoretical exploration and the application of advanced computational resources. In 2014, researchers combined Google's supercomputing power, pattern database, and group theory to establish the "God's number" — the maximum steps required to solve the cube from any given valid state — at 20 steps in the half-turn metric and 26 in the quarter-turn metric. [RKDD14] [KC07] Nowadays, computing an optimal solution for a cube state still takes hours to days, requiring extensive memory for storing heuristic tables.

1.3 Problem Statement

This thesis is dedicated to the development of efficient algorithms for solving the Rubik's Cube, with a specific emphasis on balancing solution quality against computational constraints. The first goal is to create a validation algorithm using group theory principles to verify whether a given state of the Rubik's Cube can be solved using allowed moves.

Furthermore, this study aims to integrate group theory and deep learning to develop enhanced heuristic algorithms. These algorithms are designed to find suboptimal solutions under two distinct constraints: solving the cube as fast as possible with a target of no more than 26 moves, in line with "God's number" [KC07], and finding the shortest possible solution within a maximum computational time of 60 seconds.

The culmination of this research will involve implementing conceptual algorithms (validation algorithm and heuristic search algorithms) into a comprehensive solver. The performance and efficiency of these algorithms will be thoroughly analyzed through a comparative study, providing a clear insight into their effectiveness in solving the Rubik's Cube within the stated constraints.

1.4 Literature Review

Initially, human-centric methods such as the Layer by Layer (LBL) technique offered a straightforward approach, enabling beginners to grasp the basics of cube solving through a methodical, step-by-step process. Building on this, the CFOP method, developed in the 1980s by Jessica Fridrich, revolutionized human solving techniques by segmenting the solve into four distinct stages—Cross, F2L, OLL, and PLL. This method significantly reduced the number of moves required and became a cornerstone technique for speedcubing enthusiasts. The top cube solvers can solve the cube in around 50 steps and in 10 seconds.

In 1981, Thistlethwaite's algorithm revolutionized the computational solution of the Rubik's Cube by introducing a method that breaks the solution process into four stages based on four subgroups of the Rubik's Cube group, achieving the cube's first significant proof of computational solvability through stage-wise reduction and iterative deepening depth-first search (IDDFS). [Thi81] This milestone was further built upon in 1997 by Richard Korf, who improved solution efficiency by extending the IDA* search algorithm with a pattern database that precomputes lower bounds. However, the database required storage around 182 GB. [Kor97] The field experienced another leap in 2014 with Herbert Kociemba's algorithm, which set a new standard by demonstrating that the cube could be solved within 20 moves using the half-turn metric or 26 moves with the quarter-turn metric, an achievement that aligns with the optimal solution frontier known as "God's Number." This algorithm employs a two-phase approach and leverages a reference database for efficient search. [RKDD14]

The advent of **DeepCube** in 2018 marked a revolutionary phase where machine learning techniques, specifically Monte Carlo Tree Search (MCTS) and deep reinforcement learning, were applied to solve the Rubik's Cube. [MASB18] This paper describes a novel training scheme using the Autodidactic Iteration policy to estimate the best move under a certain cube state and probability distribution on each move on doing this move can reach the goal state. By implementing MCTS as the inference algorithm based on the resulting model, this method can find the optimal solution 74% of the time in a scrambling length of 15. This approach not only demonstrated the capability to solve the cube from any scrambled state but also edged closer to achieving optimality in the number of moves.

In 2019, Forest Agostinelli and his colleagues introduced **DeepCubeA**, a novel machine-learning algorithm designed to surpass the performance of its predecessor, DeepCube. [AMSB19] This advanced algorithm utilized deep approximate value iteration within a reinforcement learning framework to precisely estimate the minimal number of steps required to solve the Rubik's Cube. Coupled with an A* search strategy and implemented using C++, DeepCubeA significantly improved both the speed and quality of solutions. In testing with 1,000 randomly scrambled cubes, each with a scramble length of 1,000 moves, DeepCubeA achieved an optimal solution in 60.3% of cases, exploring an average of 6.62 million nodes and taking about 61.14 seconds per solve. Building upon this progress, Kyo Takano in May 2023 presented a self-supervised learning model that innovatively predicts the last move in a cube's scramble sequence. [Tak23] By effectively "un-scrambling" the cube and employing a beam search with a width of 2^7, Takano's model outperformed DeepCubeA in efficiency and efficacy. It not only reduced the node exploration rate to approximately 4.18 million but also increased the rate of finding the optimal solution to around 70%, marking a significant leap forward in the application of machine learning to solve the Rubik's Cube.

This thesis integrates the methodologies and insights of Forest Agostinelli et al and Kyo Takano, combining deep reinforcement learning with deep supervised learning approaches to forge a new heuristic function. This innovative function seeks to enhance the capabilities of informed search algorithms in the quest to solve the Rubik's Cube, marking a significant milestone in the continuous development of cube-solving techniques. Additionally, by incorporating advancements from group theory, we delve deeper into crafting more efficient suboptimal algorithms tailored to achieve the predefined objectives for solving the Rubik's Cube.

1.5 Structure of the Thesis

This thesis is structured to guide the reader through the conceptual foundation, methodology, experimental validation, and implications of the proposed approach.

The preliminary chapter (Chapter 2) lays the groundwork by presenting the fundamental concepts of group theory, heuristic search algorithms, and deep learning. It prepares the reader for subsequent discussions on the varied methodologies and topics that will be explored throughout this thesis.

In the next chapter (Chapter 3) we dive into the Rubik's Cube group, exploring its significance and application in developing and enhancing algorithms for validating cube states and solving the Rubik's Cube. This section serves as a bridge between theoretical mathematical foundations and practical solving strategies.

To do heuristic searches, creating a heuristic function for guidance is essential. Chapter 4 explores the training of models using deep reinforcement learning and supervised learning to predict the proximity of a given cube state to the solved state. It also discusses leveraging computer hardware, such as GPUs, for batch processing to enhance the efficiency of training, processing, and searching.

By leveraging the models created by Chapter 4 and prevention algorithm from Chapter 3, Chapter 5 integrates and implements the search algorithms subject to various purposes mentioned in Section 1.3. This chapter demonstrates the modification and improvements done to the classical heuristic search algorithms like A* search and beam search to gain better performance and efficiency.

With the implementations of search algorithms, Chapter 6 presents a comprehensive analysis of the algorithms' efficiency through empirical experiments conducted with extensive datasets. It evaluates the performance of the designed and implemented algorithms, offering insights into their practical implications.

Finally, Chapter 7 discusses the advantages and disadvantages of each algorithm, providing a critical evaluation of their practical applications and theoretical implications. It also outlines future directions, such as the exploration of new heuristic functions and their application to more complex puzzles.

Chapter 2

Preliminary

2.1 Understanding the Rubik's Cube

Before diving deep into the intricacies of the Rubik's Cube, it's imperative to familiarize ourselves with some foundational terms that are commonly used. These terms serve as the building blocks for understanding more complex concepts and techniques related to the cube.

Definition 2.1.1. ***Facets***: The colored sub-faces on the surface of the cube. There are 54 total facets. One can imagine facets as the stickers on the Rubik's cube.

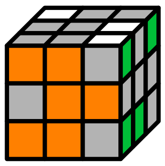


Figure 2.1: Examples of some facets of the Rubik's cube.

Cubies: The distinct small cubes that together make up the Rubik's Cube. These components come in three varieties: **Center Cubies**, **Edge Cubies**, and **Corner Cubies**. Center cubies are the fixed, central pieces on each face, determining the color of that face. Edge cubies have two colored facets and occupy the positions between the center and corner cubies. These cubies are located at the edges of the cube. Similarly, corner cubies have three colored facets and are located at the corners of the cube. Examples of cubies can be seen in Fig. 2.2.

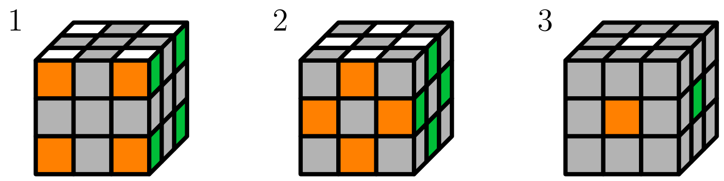


Figure 2.2: The illustration of corner cubies (1), edge cubies (2), and center cubies(3).

Face: One of the six sides of the Rubik's Cube. Each face consists of 9 facets. When the cube is solved, each face corresponds to a unique color shown in Fig. 2.3.

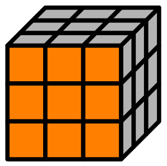


Figure 2.3: The F (Front) face of the Rubik's Cube.

Layer: A single, complete row of cubies on the Rubik's Cube, either horizontal or vertical shown in Definition 2.1.1. Each layer consists of 21 facets, and 9 cubies. Each layer can be independently rotated, in fact, these moves generate all the allowed operations on the cube as we will see.

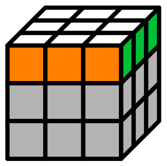


Figure 2.4: The U (Up) layer of the Rubik's Cube.

Definition 2.1.2. Moves Allowed: There are six types of layer rotations, each aligning with the rotation of one of its faces. By Singmaster's notation, which is a standardized method for documenting cube moves, we use the symbols `U, D, R, L, F` and `B` to signify a 90-degree clockwise rotation of the Upper, Down, Right, Left, Front, and Back layers, respectively. For example, `U` denotes a 90-degree clockwise turn of the up layer.

To represent anticlockwise rotations, we append a prime symbol ($'$) to the letter of the corresponding face turn, such as `U'` for a 90-degree anticlockwise rotation of the up layer.

In this thesis, we focus on clockwise and anticlockwise turns, which we denote as **quarter-metric** turning. This specification leads to a total of 12 allowed operations or rotations on the Rubik's Cube, including both the clockwise and anticlockwise directions for each of the six faces. All the moves can be seen in Fig. 2.5.

We denote the set of all allowed single moves as Ψ where

$$\Psi = \{\texttt{R, R', L, L', U, U', D, D', F, F', B, B'}\}$$

To make it easier for the readers, we standardize the orientation by designating orange as the front face (`F`) and white as the up face (`U`) shown in Fig. 1.1. While individual solvers might prefer different orientations for the front and up faces, it's important to note that the choice of orientation does not alter the encoding scheme or the representation of the cube's state. The decision to standardize on orange and white is purely for clarity and ease of communication; any cube orientation follows the same principles and can be adapted to this framework without affecting the underlying methodology. Essentially, the orientation selected is a matter of personal preference and does not impact the cube's solution strategy or analysis.

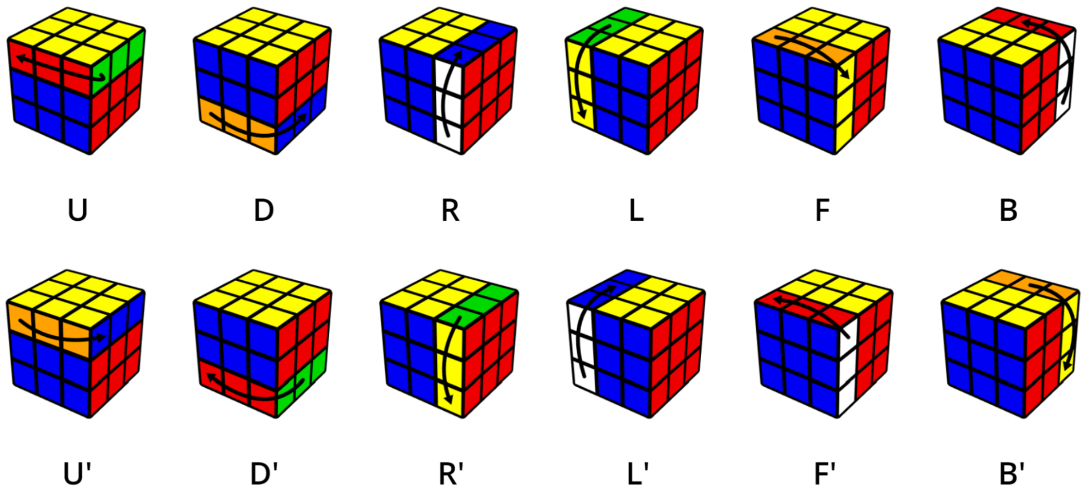


Figure 2.5: Allowable moves for Rubik's Cube and their notations

2.2 Theoretical Foundation of Group Theory

Group theory is a branch of mathematics that studies algebraic structures. This plays a necessary role in deciphering the complexities inherent in the Rubik's Cube. In this section, we will outline some important results and theorems in group theory, which are essential for understanding the analyses presented in the ensuing chapter (Chapter 3). For a more detailed and substantial introduction, see [Lee18] and [DF04].

2.2.1 Groups and Subgroup

First, we need to define groups and subgroups.

Definition 2.2.1. A ***group*** is a nonempty set G, together with a binary operation $*$, satisfying the following conditions:

1. $a * b \in G$ for all $a, b \in G$ (closure);
2. $(a * b) * c = a * (b * c)$ for all $a, b, c \in G$ (associativity);
3. there exists an $e \in G$ such that $a * e = e * a = a$ for all $a \in G$ (existence of identity); and
4. for each $a \in G$, there exists a $b \in G$ such that $a * b = b * a = e$ (existence of inverses).

Two special types of groups are essential for our understanding of the following proofs and explorations.

Definition 2.2.2. A group G is said to be ***abelian*** if $a * b = b * a$ for all $a, b \in G$. In other words, the operation of G is commutative. A group G becomes a ***cyclic*** group if there exists an element $a \in G$, such that every element of G is in the form

a^n, where n is any integer. The element a is called the generator of G and G is called a cyclic group generated by a and is denoted by $G = \langle a \rangle$.

Example 2.2.3. For example, consider the set of integers $\mathbb{Z}$ with the binary addition operation $+$. We denote this set as $\mathbb{Z}^+ = (\mathbb{Z}, +)$. The addition between integers will always result in an integer, therefore $\mathbb{Z}^+$ is close. Moreover, for any integers, adding a zero doesn't change it, so 0 is the identity for $\mathbb{Z}^+$. Moreover, for every integer $z \in \mathbb{Z}$, there must exist an integer $-z$ such that $z + (-z) = (-z) + z = 0$. So, $\mathbb{Z}^+$ satisfies the existence of the inverses as well. Thus, $\mathbb{Z}^+$ is a group.

Since addition is commutative, so $\mathbb{Z}^+$ is an abelian group as $(a + b) = (b + a)$. Moreover, the $\mathbb{Z}^+$ is a cyclic group generated by 1 and -1, because every integer can be expressed as a sum involving $1, -1$, or both.

However, if we change the binary operation to multiplication, then the existence of the inverses axiom will fail as we are working in the set of integers.

One of the most crucial ways of obtaining new groups or extracting targeted groups from the original groups is to construct a subgroup from the original group. The concept of subgroup gives us a way to divide the data into different subsets with similar properties.

Definition 2.2.4. Let G be a group with operation $*$. Then a subset H of G is called a ***subgroup*** of G if H is a group under the same operation $*$. In this case, we write $H \leq G$. Note that every group is a subgroup of itself, and $\{e\}$ is also a subgroup of every group.

For example, the set of all even integers $\mathbb{E}$ under the operation of addition is a subgroup of the group of integers $\mathbb{Z}^+$ because it satisfies all the necessary group axioms under addition. Firstly, closure is satisfied because the sum of any two even integers remains even. Secondly, associativity is inherent as the operation of addition among integers is associative. The identity element is present as the number 0, which is even and serves as the identity element in $\mathbb{E}$. Additionally, every even integer has an inverse within $\mathbb{E}$; specifically, for any even integer a, the integer $-a$ is also even and acts as its inverse since $a + (-a) = 0$. Therefore, since $\mathbb{E}$ satisfies all the group axioms under the same operation as $\mathbb{Z}^+$, it is a subgroup of $\mathbb{Z}^+$, denoted as $\mathbb{E} \leq G$.

Sometimes we are curious about the number of elements in a group, such as in the group associated with the Rubik's Cube. To accurately describe the number of configurations in the Rubik's Cube group, we use the concept of 'order'. The order of a group represents the total number of its elements, providing a fundamental metric for understanding its structure and complexity.

Definition 2.2.5. Let G be a group. Then, the ***order*** of G, denoted as $|G|$, is the cardinality of G, i.e., the number of the elements in the set G. If $|G|$ is finite, then G is a finite group; otherwise, it's an infinite group.

The concept of order also applies to individual elements within a group, reflecting the element's cyclic properties. The **order** of a for any $a \in G$, denoted by $|a|$, is the smallest positive integer n such that $a^n = e$, where e is the identity element of G. If such an n exists, a is said to have finite order. If no such n exists, then a is said to have infinite order.

Example 2.2.6. Consider the group $\mathbb{Z}_6 = \{0, 1, 2, 3, 4, 5\}$ under addition modulo 6, which has an order of 6, indicating it is a finite group. We write $|\mathbb{Z}_6| = 6$. In the same group, the element 2 has an order of 3 because 2 added to itself three times modulo 6 returns zero, the identity element. We write $|2| = 3$.

2.2.2 Symmetric Group

The other type of group that is essential for our analysis of the Rubik's cube is the permutation group. If A is a set and consider the set of permutations $\{f : A \to A \mid f \text{ is bijective}\}$ with the operation of composition. One can check that the set of permutations with the composition forms a group.

The symmetric group, denoted as S_n, represents the group of all possible permutations of a set of n elements. Formally, the symmetric group S_n and the permutations in S_n are defined as:

Definition 2.2.7. The symmetric group of degree n, S_n, is the set of all permutations of the set $1, 2, \ldots, n$ for a positive integer n, equipped with the operation of permutation composition. Let S_n be the symmetric group on n symbols, and let k be a positive integer such that $1 \leq k \leq n$. A k-cycle in S_n is a permutation σ of the form $\sigma = (a_1\ a_2\ \ldots\ a_k)$ where:

1. Each a_i for $i = 1, 2, \ldots, k$ is a distinct element of the set $1, 2, \ldots, n$;
2. $\sigma(a_i) = a_{i+1}$ for $i = 1, 2, \ldots, k-1$;
3. $\sigma(a_k) = a_1$;
4. for any $x \notin \{a_1, a_2, \ldots, a_k\}$, $\sigma(x) = x$.

In other words, a k-cycle is a permutation that rotates k specific elements of the set among themselves in a specific cyclic order and leaves all other elements of the set unchanged. The notation $(a_1\ a_2\ \ldots\ a_k)$ denotes that a_1 is mapped to a_2, a_2 is mapped to a_3, and so on, with a_k being mapped back to a_1.

Remark 2.2.8. For any k-cycles $\sigma, \tau \in S_n$ and $i \in \{1, \ldots, n\}$, suppose we want to check which element in $\{1, \ldots, n\}$ is i been sent to by $\sigma\tau$. Then we trace it through the element taking the cycles from right to left but within each cycle working left-to-right. So, we start to trace i in τ and suppose τ sends i to j, then we trace j in σ.

And suppose $\sigma = (1\ 2\ \cdots\ n)$, then the inverse of σ, σ^{-1} is just sending each element with its reversed order of σ, i.e., $\sigma^{-1} = (n\ n-1\ \cdots\ 2\ 1)$.

Remark 2.2.9. For any $\sigma \in S_n$, it can be directly written out using cycle notation. For example, $(1\ 5\ 8\ 4) \in S_8$ can be written as

$$(1\ 5\ 8\ 4) = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 5 & 2 & 3 & 1 & 8 & 6 & 7 & 4 \end{pmatrix}$$

where each number in the first row represents an element from the set $\{1, 2, \ldots, n\}$, and the corresponding element in the second row indicates where the element from the first row is sent under the permutation σ. The cycle notation $(1\ 5\ 8\ 4)$ indicates that 1 is sent to 5, 5 to 8, 8 to 4, and 4 back to 1, while elements not listed in the cycle (2, 3, 6, 7) are fixed, meaning they map to themselves. And we can write the permutation from the cycle notation reversely using a similar procedure.

Example 2.2.10. Let's explore the symmetric group S_3, which consists of all permutations of the set $1, 2, 3$. This will provide a concrete example of how permutations work and how they form a group under composition.

Consider a permutation σ where $\sigma(1) = 2$, $\sigma(2) = 1$, and $\sigma(3) = 3$. This can be written in cycle notation as $(1\ 2)$ as a 2-cycle. It swaps the positions of 1 and 2, leaving 3 unchanged. Take another permutation τ where $\tau(1) = 3$, $\tau(2) = 1$, and $\tau(3) = 2$, represented as $(1\ 3\ 2)$ in cycle notation. It cyclically permutes 1 to 3, 3 to 2, and 2 to 1.

Then, the composition of σ and τ, denoted $\sigma \circ \tau$ or $\sigma\tau$, results in a new permutation where each element is first permuted by τ and then by σ. 1 is being sent to 3 in τ and 3 is not in σ, which means 3 is sending back to 1 again. Moreover, 2 gets sent to 1 in τ and 1 is sent back to 2 in σ. Therefore, $\sigma\tau = (1\ 3)(2) = (1\ 3)$.

It's clear that the order of S_n is $n!$ for any positive integer n, i.e., $|S_n| = n!$ for $n > 0 \in \mathbb{Z}$. Since the number of permutations of n objects is $n!$ and S_n encompass all the permutations of n objects, so $|S_n| = n!$.

Definition 2.2.11. A **transposition** is a 2-cycle. Let σ be a permutation of a finite set of n elements, i.e., $\sigma \in S_n$. The **sign** of σ, denoted $\operatorname{sgn}(\sigma)$, is defined as 1 if σ can be expressed as a product of an even number of transpositions, and -1 if σ can be expressed as a product of an odd number of transpositions. We'll see shortly that sgn is well-defined.

Consider the permutation $\sigma = (1\ 2\ 3)$ in S_3, this permutation can be decomposed into two transpositions: $(1\ 2\ 3) = (1\ 2)(2\ 3)$. Since there are two transpositions, an even number, the sign of σ is $+1$, making it an even permutation.

Theorem 2.2.12. *Let $\sigma \in S_n$ be a k-cycle, then*

1. *$sgn(\sigma)$ is well-defined.* [DF04] *Hence, let $\sigma' \in S_n$, then $sgn(\sigma\sigma') = sgn(\sigma) \cdot sgn(\sigma')$.*

2. *σ can be written as a product of $k - 1$ transpositions.* [DF04] *Therefore, the sign of σ, denoted $sgn(\sigma)$, is given by:*

$$sgn(\sigma) = \begin{cases} 1 & \text{if } k \text{ is odd,} \\ -1 & \text{if } k \text{ is even.} \end{cases}$$

Thus, we can express the sign of σ as:

$$sgn(\sigma) = (-1)^{k-1}$$

.

Example 2.2.13. As seen before, the 3-cycle $\sigma = (1\ 2\ 3)$ can be decomposed into the product of 2 transpositions. Therefore by Theorem 2.2.12, $\operatorname{sgn}(\sigma) = 1$ since $\operatorname{sgn}(\sigma) = (-1)^2 = 1$. Consider $\tau = (1\ 4\ 3)$ and $\pi = (2\ 6)$, then by Theorem 2.2.12, $\operatorname{sgn}(\tau\pi) = 1 \cdot (-1) = -1$. We can verify it by computing $\tau\pi$ as the product of transpositions of $(1\ 4)(4\ 3)(2\ 6)$ which is odd permutation, hence it has sign -1.

Definition 2.2.14. The ***alternating group*** A_n is the set of all even permutations in S_n. If $n \geq 2$, then $|A_n|$ is exactly half of $|S_n|$. [Lee18] One can check that A_n is a subgroup of S_n.

Theorem 2.2.15. *For any $n \geq 5$, A_n is generated by the 3-cycles. That is if these 3-cycles are in a subgroup of A_n, then the subgroup is A_n.*

$$(1\ 2\ 3), (1\ 2\ 4), \ldots, (1\ 2\ n)$$

Proof of Theorem 2.2.15. Before starting the proof, we list some equations between permutations that are essential to our proof.[Ban82, DF04] Note that a, b, c, d are distinct elements of $\{1, 2, \ldots, n\}$ and x, y, z are distinct elements of $\{3, 4, \ldots, n\}$

1. $(a\ b)(c\ d) = (c\ a\ d)(a\ b\ c)$
2. $(a\ b)(a\ c) = (a\ c\ b)$
3. $(x\ y\ z) = (1\ 2\ x)(2\ y\ z)(1\ 2\ x)^{-1}$
4. $(2\ y\ z) = (1\ 2\ y)(1\ 2\ z)(1\ 2\ y)^{-1}$
5. $(1\ y\ z) = (1\ 2\ z)^{-1}(1\ 2\ y)(1\ 2\ z)$

One can check simply by multiplying out the permutations and checking how each permutation sends elements using the operation order mentioned in Remark 2.2.8. For example, $(1\ 2\ x)(2\ y\ z)(1\ 2\ x)^{-1}$ sends x firstly to 2 and then to y by Remark 2.2.8. It sends y to z and z to 2 then to x. Therefore, it is exactly $(x\ y\ z)$.

Now let's start with the proof. The first step is to show that A_n is generated by 3-cycles. By Definition 2.2.14, all elements in A_n are even permutations and can be written as products of even numbers of transpositions. Each element can therefore be written as a product of elements, each of which is the product of a pair of transpositions. We assume that the two transpositions in any pair are distinct, otherwise, their product is the identity. Hence, any element of A_n is a product of elements, each either of the form $(a\ b)(c\ d)$ or the form $(a\ b)(a\ c)$ where a, b, c, d are distinct. Since each form can be written as a product of or one 3-cycle based on Statements 1 and 2, it follows 3-cycles generate that A_n.

Let $H = \langle (1\ 2\ 3), (1\ 2\ 4), \ldots, (1\ 2\ n) \rangle$, that is the smallest subgroup of S_n containing all these 3-cycles. This is the smallest group generated by a set of all 3-cycles. We want to show that $H = A_n$. Note that each $i \in H$ is a 3-cycle, so $i \in A_n$ as it is an even permutation. Clearly, $H \leq A_n$. We will focus on showing $A_n = H$. By Statements 4 and 5, each 3-cycle of the form $(1\ a\ b)$ or $(2\ a\ b)$ with $a, b \geq 3$ is in H. Then by Statement 3, every 3-cycle is in the H. Since we proved that A_n is generated by 3-cycle, so A_n is generated by $(1\ 2\ 3), (1\ 2\ 4), \ldots, (1\ 2\ n)$. We have that $H = A_n$. □

2.3 Heuristic Search Algorithms

The main part of this thesis is to design heuristic search algorithms. In this section, we delve into the fundamental concepts and methodologies of the heuristic search algorithms.

The cornerstone of the heuristic search algorithms is the heuristic function which estimates the distance from the current state to the goal state. Unlike blind search strategies such as Breadth-First Search (BFS) or Depth-First Search (DFS), which aimlessly navigate through the solution space, heuristic search algorithms employ this function to prioritize certain paths over others based on the heuristic function. This ensures a more directed and efficient approach to finding a solution in a vast search space.

2.3.1 Heuristic Functions

A **heuristic function**, h, offers a best-guess estimate of the solution to a problem when identifying the optimal solution is impractical, typically due to constraints in time or computational resources. Heuristic methods facilitate faster problem-solving by providing approximate solutions where traditional methods may be too slow or fail to yield an exact answer. For example, a well-known heuristic function in path-finding problems is the Manhattan distance, used in grid-based search scenarios.

Suppose that the search starts at point $s = (s_x, s_y)$ to find a goal at $g = (g_x, g_y)$. Then, for any points $t = (t_x, t_y)$ in the grid, the Manhattan distance is defined as

$$h(t, g) = |t_x - g_x| + |t_y - g_y|$$

which is visualized in Fig. 2.6. This function estimates the total "grid distance" an agent must traverse from t to g along the grid's axes to move from the starting point directly to the goal, without diagonal movement.

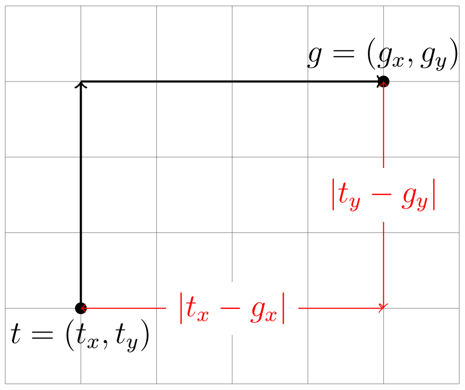


Figure 2.6: Visualization of Manhattan distance in a grid. The red arrows show the horizontal and vertical distances that sum up to give the Manhattan distance.

Another example is the pattern database where the solution space of a problem is precomputed and stored in a database. This approach is used in Korf's paper which demonstrates its effectiveness. [Kor97] During the search, the heuristic function queries this database to obtain an accurate estimate of the minimum number of moves required from the current state. Pattern Database Heuristics are particularly powerful in complex problem spaces, such as the Rubik's cube, where they can drastically reduce the search space and time by leveraging the precomputed solutions. However, one disadvantage is that the pattern database may take a huge amount of memory if the search space is vast.

2.3.2 Properties of Heuristic Functions

When developing a heuristic function, there are two essential properties to aim for: **admissibility** and **consistency**.

A heuristic is considered admissible if it consistently underestimates or precisely estimates—never overestimates—the cost of reaching the goal state from any given state. Formally, let g be the goal state and n be any state in the search space. Suppose h^* is the optimal cost-to-go function that can precisely measure the minimum distance between g and n. Then a heuristic function h is admissible if

$$h(n, g) \leq h^*(n, g)$$

where $h(n, g)$ is the estimated cost from n to g. This characteristic is important to ensure that the solution proposed by algorithms, such as A* search, is indeed the optimal one.

On the other hand, a heuristic is deemed **consistent** (or monotonic) if, for every node and its successor, the estimated cost from the initial state to the goal via the successor is not greater than the cost from the initial state to the current node plus the cost of reaching the successor from the current node. Formally, let n be any state in the search space, n_1 be any one of its neighbors, and $c(n, n_1)$ be the cost of transitioning from n to n_1, then a heuristic function h is said to be consistent if for all n in the search space

$$h(n, g) \leq c(n, n_1) + h(n_1, g)$$

This property is critical as it guarantees that once a node's cost is calculated, there is no need to reassess it, thus optimizing the efficiency of the search process.

For example, the Manhattan distance heuristic function is both admissible and consistent. Firstly, in grid navigation, each move is either horizontal or vertical, moving the agent one unit closer to the goal. Therefore, the Manhattan distance never exceeds the actual minimal path cost, which means that $h(n, g) \leq h^*(n, g)$ for all n. Hence, h is admissible. In the grid, the cost $c(n, m)$ to move from n to m, is equal to 1. Considering the Manhattan distance, moving from n to an adjacent node m changes either n_x or n_y by exactly 1 unit, which implies:

$$|h(n, g) - h(m, g)| = 1, \tag{2.3.1}$$

exactly equal to the cost $c(n, m)$. This relationship upholds the consistency condition:

$$h(n, g) \leq h(m, g) + 1. \tag{2.3.2}$$

These properties ensure that the Manhattan distance heuristic not only provides a reliable lower bound on the distance to the goal but also adheres to the stepwise path cost increases required for optimal pathfinding in algorithms such as A*.

While admissibility and consistency are foundational, the real impact of a heuristic function in heuristic search algorithms lies in its ability to accurately and efficiently guide the search process toward the goal. The ultimate objective is to devise a heuristic that balances aspects like accuracy, efficiency, informativeness, and practicality to navigate complex problem spaces effectively.

In the sections that follow, including in Chapter 4, we will focus on developing a heuristic function. This function will measure how close the current state of a Rubik's Cube is to its solved state.

2.3.3 Classic Heuristic Search Algorithms

In light of the definition and properties of heuristic functions, it becomes essential to pair these functions with an effective search algorithm to fully harness their potential. Therefore, in this section, we introduce several important heuristic search algorithms.

Greedy Best-First-Search

Greedy Best-First Search (GBFS) prioritizes nodes based on their proximity to the goal state, as estimated by a heuristic function. This approach focuses solely on the heuristic value $h(n)$, ignoring the cost to reach a node. This approach abandons the solution quality and focuses on solving speed. The pseudocode can be found in Appendix.A. (Algorithm 13).

GBFS navigates towards the goal based on a heuristic, aiming for the nearest solution at each step. However, its efficiency depends heavily on the heuristic's accuracy. An accurate heuristic can lead to quick and efficient searches, while a poorly chosen one may result in extensive and unproductive exploration. GBFS's primary limitations stem from its potential to enter infinite loops or hit dead-ends in vast search spaces without a mechanism to avoid or recognize previously visited nodes. This makes GBFS memory-efficient, as it tracks only the current path and its direct alternatives, but this comes at the risk of missing solutions or pursuing suboptimal paths due to its greedy nature.

In the initial experiments, GBFS's performance has demonstrated considerable variability; it occasionally veers off course, focusing on a path that may lead to perpetual searches. This tendency towards short-term promising paths, which may culminate in dead-ends or loops, underscores the algorithm's inherent greediness. The absence of supplementary measures to catalog visited nodes exacerbates this issue, restricting the algorithm's ability to extricate itself from such predicaments.

Given its propensity for non-optimality and the risk of incompleteness—particularly in scenarios demanding exhaustive search capabilities—GBFS is deemed most appropriate for scenarios where the search space is manageable, and precision in the path's optimality is not paramount. Therefore, this thesis avoids GBFS in favor of alternative strategies better suited to tackle the expansive search space and find solutions with a certain quality level quickly.

Beam Search

Beam Search is a variant of the best-first search that limits the number of nodes retained at each level of the search tree to a predetermined number, known as the beam width. This approach helps manage memory usage and computation time. The conceptual pseudocode can be seen in Algorithm 1. Moreover, a demonstration of the beam search can be visualized in Fig. 2.7 which provides a more intuitive approach to beam search. In the visualization, in each depth, we only store the best N nodes regarding the beam width and expand these N nodes in the next depth. For all other nodes, we leave them out permanently from our search.

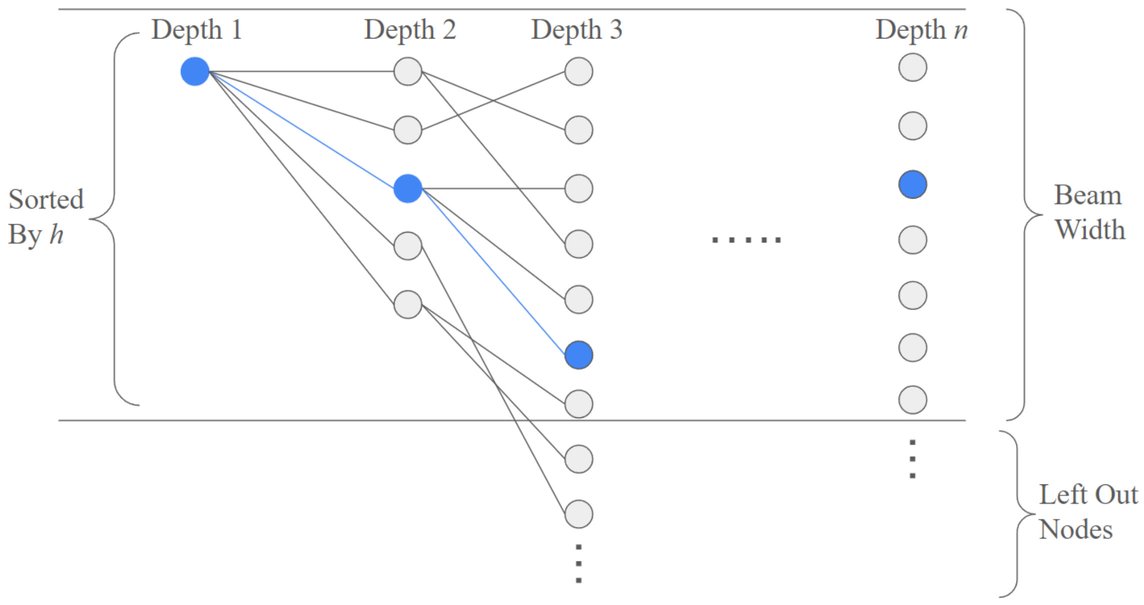


Figure 2.7: Visualization of Beam Search

Notice the difference between Beam Search and GBFS, Beam Search expands nodes level by level, maintaining a balance between breadth-first and depth-first characteristics by focusing on a narrower set of nodes at each depth. Unlike Beam Search, GBFS always selects the single most promising node to expand next throughout the search, based on a heuristic estimate of the cost from the node to the goal. This selection is global and does not limit the number of nodes to retain at each level.

Beam search is known for its memory efficiency, as it maintains only a limited set of options at any given time. Consider a search scenario with depth N, beam size B, and each node having b neighbors. The maximum number of nodes explored during a search would be $N \cdot B \cdot b$. This configuration leads to a linear computational complexity relative to the number of expansions, which is advantageous, particularly in large or infinite state spaces where it is impractical to store all nodes.

However, beam search shares a limitation with GBFS, which relies solely on a heuristic function h to guide its decisions. This dependency can lead to problems if the

Algorithm 1: Beam Search

```
Data: $x_0$: Initially Scrambled Cube, $BeamSize$
Result: Solution or Failure.
1  $Beam \leftarrow \{x_0\}$;
2  while $Beam \neq \emptyset$ do
       /* Expand Node in Beam */
3      for $node \in Beam$ do
4          Expand $node$;
5      Evaluate the heuristic value $h(n)$ of each successor;
6      $BestNodes \leftarrow \{ \}$;
       /* Get the Best $BeamSize$ Nodes */
7      while $size(BestNodes) \leq BeamSize$ do
8          Pop the best node $n$ from $successors$ to $BestNodes$ based on $h(n)$;
9          if $n$ is solved then
10             return $n$;
11     $Beam \leftarrow BestNodes$;
12 return Search Failed;
```

heuristic is poorly designed, such as looping or premature convergence on suboptimal solutions. The method is particularly susceptible to getting stuck in local optima due to its limited exploration, which may not be sufficient to find escape paths to better solutions. Furthermore, beam search is not complete, meaning it does not guarantee a solution even if one exists, especially if the beam size is small and the search depth is shallow. Thus, careful tuning of the beam size and search depth is crucial to ensure a thorough and effective exploration of the search space.

A* Search

For any node n in the search space, A* search combines the costs to reach the node n from the starting node, $g(n)$, and the estimated cost from the node n to the goal $h(n)$ produced by the heuristic function to form the fitness of the node n, $f(n) = g(n) + h(n)$. The primary metric or guidance for A* search is $f(n)$ instead of $h(n)$ compared to the last two search algorithms. It efficiently finds the lowest cost path by exploring paths that, in the aggregate, are promising. The pseudocode can be found in Algorithm 2.

The A* search algorithm is renowned for its capability to provide a guaranteed optimal solution under the appropriate conditions, showcasing its adaptability across a diverse array of problem scenarios. [DP85] This adaptability is further enhanced by its use of heuristics, which guide the search efficiently and enhance processing speed in numerous practical applications like motion planning. [YLX+10] The key to its success lies in its hybrid approach: A* combines the strategies of Dijkstra's Algorithm, which prioritizes vertices close to the starting point, with those of Greedy Best-First Search, which favors vertices near the goal. This blend allows A* to efficiently navigate

Algorithm 2: A* Search

```
Data: $x_0$: Initially Scrambled Cube
Result: Solution or Failure.
1  $OpenList \leftarrow \{x_0\}$;
2  $CloseList \leftarrow \{\}$;
3  while $OpenList \neq \emptyset$ do
4      $Node \leftarrow$ The node with the lowest $f(n)$ from $OpenList$;
5      if $Node$ is solved then
6          return $Node$
7      $successors \leftarrow$ Expand $Node$;
8      Evaluate the heuristic value $h(n)$ of each successor;
9      for $successor \in successors$ do
10         $f(successor) = g(x_0, successor) + h(successor)$;
11         if $successor = node$ for some $node \in OpenList$ then
12             if $node.f \leq successor.f$ then
13                 $Continue$ ;                       // Since the old one is better
14         if $successor = node$ for some $node \in CloseList$ then
15             if $node.f \leq successor.f$ then
16                 $Continue$;
17             else
                   /* We are re-exploring the node with lower $f(n)$            */
18                 $CloseList \leftarrow OpenList \setminus \{node\}$;
19         $OpenList \leftarrow OpenList \cup \{successor\}$;
20     $CloseList \leftarrow CloseList \cup \{Node\}$;
21 return Search Failed;
```

through the search space by optimally balancing the exploration and exploitation ratio.

However, A* also has its drawbacks; notably, it can consume a substantial amount of memory due to the necessity to store all explored paths. Additionally, the effectiveness of A* heavily depends on the quality of the heuristic employed. If the heuristic is poorly chosen or if there are constraints on memory availability, the performance of the A* algorithm can significantly degrade.

In contrast, beam search, while capable of handling larger search spaces by limiting the number of paths retained, may compromise on finding the most optimal solution, potentially overlooking the best outcome in favor of broader search capabilities.

Iterative deepening A* Search (IDA* Search)

IDA* (Iterative Deepening A*) algorithm for solving the Rubik's cube, first introduced by Korf in 1997 [Kor97], marked a significant advancement in heuristic search algorithms applied to the Rubik's Cube. Utilizing a pattern database as the heuristic function, IDA* incorporates group theory knowledge to guide its search, ensuring an optimal solution under the right conditions.

IDA* effectively combines the space efficiency of depth-first search with the heuristic guidance characteristic of A*. The algorithm operates by iteratively deepening the search depth based on the cost function $f(n)$, exploring only those nodes that fall within a dynamically adjusted cost threshold. This method allows IDA* to maintain a manageable memory footprint while systematically broadening the search scope. The pseudocode for IDA*, while not explored in detail within this thesis, is provided in Appendix A for Algorithm 14.

Despite its historical significance and effectiveness, the IDA* algorithm was not chosen as a focus for this thesis due to several constraints related to its operation. Primarily, IDA*'s performance is highly dependent on the admissibility and consistency of the heuristic function $h(n)$. An inadmissible heuristic can lead to the algorithm becoming ensnared in suboptimal branches or loops, significantly detracting from its efficiency.

In practical applications and initial testing, IDA* performs exceptionally well when the scrambling length is short, where the heuristic function closely approximates the true cost-to-go, ensuring swift and accurate solutions. However, as the scrambling length increases, the precision of $h(n)$ tends to diminish, leading to increased computational idling and slow progression in expanding the search threshold. This sensitivity to heuristic accuracy, coupled with challenges in guaranteeing heuristic admissibility in heuristic functions approximated via deep learning models, led to the decision to exclude IDA* from the primary investigative framework of this thesis.

2.4 Fundamentals of Deep Learning

Deep learning plays an important role in this thesis, primarily because our ultimate objective is to develop a heuristic function that estimates the proximity from the current cube state to the solved state using deep learning. This function is crucial for guiding heuristic searches aimed at solving the Rubik's Cube from any valid starting state. To achieve this, it is essential to first establish a solid understanding of the deep learning framework.

2.4.1 Introduction to Machine Learning

At its core, machine learning is a subset of artificial intelligence (AI) focused on building systems that learn from data, identify patterns, and make decisions with minimal human intervention. The ultimate goal is to create models that can accurately predict the expected output based on the input data. These models refine their predictions by learning from errors and incorporating additional data inputs. In subsequent sections, we will explore how these models are built, trained, and optimized, and examine various techniques that enhance their ability to make increasingly accurate predictions.

Definition and Scope

Mitchell's book provides a nice generation of the scheme of machine learning algorithms: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." [GBC16]. To illustrate, consider a machine learning model designed for image recognition T, which improves its accuracy P measure by some metrics as it processes more images E. We will explore different techniques and concepts to form P and E.

In this paper, the task T received by the machine learning algorithm is that the machine is trying to learn the features and patterns regarding the Rubik's cube states. For example, one of the tasks is to predict the minimal number of steps required to solve the cube given the specific state of a cube.

Measuring the Performance of A Model

To effectively evaluate the performance of a model, it is crucial to measure the discrepancy between the expected output y and the predicted output $\hat{y}$ produced by the model for a given input x. Ideally, if $y - \hat{y} = 0$ for every x within the input space, then the loss would be zero, indicating that the model perfectly predicts the true values based on the inputs. However, achieving this level of accuracy is typically unfeasible in practical scenarios.

The learning process of the model involves optimizing a set of parameters (θ), which dictate how inputs are transformed into predictions. This optimization occurs iteratively, improving the model's parameters based on the loss calculated from previous iterations in a certain way (introduced in Section 2.4.2). During training, a batch of data is processed at once, which not only enhances computational efficiency but also helps in stabilizing the learning updates.

Given that the model updates are based on batch loss rather than individual data points, it is crucial to use appropriate metrics to quantify this loss. To this end, we introduce two commonly used loss functions suitable for batch processing: **mean square error** (MSE) for regression tasks and **cross-entropy error** (CEE) for classification tasks.

Mean Square Error (MSE) is particularly useful when the model outputs are continuous values. For example, in predicting house prices based on features like location and size, MSE quantifies prediction accuracy by averaging the squared differences between predicted and actual prices, effectively penalizing larger errors to improve model accuracy.

If we assume that each batch used for training consists of m samples, and the model predicts a vector of values $\hat{\mathbf{y}} = [\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m]$ where each $\hat{y}_i$ corresponds to the prediction for the i-th input in the batch. And the true outputs (values) are given by $\mathbf{y} = [y_1, y_2, \ldots, y_m]$, then the mean square loss is defined as Equation 2.4.1.

$$MSE = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2 \tag{2.4.1}$$

In a scenario where the model achieves perfect predictions, the MSE would be zero, indicating that for each input i, $y_i - \hat{y}_i = 0$ which leads to $MSE = 0$. Conversely, if the model's predictions significantly deviate from the actual values, then for each i, $y_i - \hat{y}_i$ will be larger which leads to the increase in MSE.

MSE serves as a crucial metric for evaluating regression tasks by quantifying how closely predicted values align with actual values across a batch of data. However, machine learning is not limited to regression tasks; it also extends to multi-class classification tasks, where the model outputs a probability distribution representing the likelihood that a given input belongs to various classes. For example, in image recognition, a model might classify images into categories such as animals, vehicles, and landscapes based on their visual content, outputting a probability distribution across these classes for each image. In such cases, MSE is unsuitable since the output is not continuous.

Instead, we employ ***Cross-Entropy Error*** (CEE) loss to measure the discrepancy between two probability distributions. For example, if there are c classes, and a given point $x \in X$, where X is the sample space, has a true probability distribution $p(x)$ among these classes (e.g., $p(x)_1 = 0.2$ indicates a 20% probability that x belongs to class 1). The model attempts to approximate this distribution with its output $q(x)$.

The error between these distributions on a single input x is quantified by Equation 2.4.2.

$$CEE = -\sum_{i=1}^{c} p(x)_i \cdot \log q(x)_i \tag{2.4.2}$$

Note that each $q(x)_i$ and $p(x)_i$ is always going to be in the range of $[0, 1]$. Therefore, $\log(q(x))$ will always be smaller than 0. So, $CEE \geq 0$. when the model perfectly predicts the true class probabilities, then for all i such that $p(x)_i = 1$, $q(x)_i = 1$. This leads to $\log q(x)_i = 0$ which further implies $CEE = 0$.

However, CEE can diverge when model predictions are poor, particularly if $q(x)_i$ approaches zero where $p(x)_i$ is significant, potentially leading to $-\log 0$, which is infinite.

To accommodate batch processing, where the model is trained on a batch of inputs of size m, we average the CEE for each point in the batch B which leads to Equation Diagram (2.4.3).

$$CEE = \frac{1}{m} \sum_{j=1}^{m} \sum_{i=1}^{c} p(B_j)_i \cdot \log q(B_j)_i \tag{2.4.3}$$

where B_j means the j-th input in B and i refers the class index among the c.

Together, equations for MSE and CEE provide metrics to gauge model performance, guiding parameter adjustments (θ) in subsequent iterations to enhance performance.

Categories of Machine Learning

The learning experience E can be broadly classified into three categories based on the nature of the learning signal, data type, and feedback available to the system:

- **Supervised Learning**: In supervised learning, the algorithm learns from a labeled dataset, provided with the correct or expected answers (output) for given inputs. The model makes predictions or decisions based on input data and is corrected when its predictions are inaccurate based on the expected output. This type of learning is used in applications such as spam detection, sentiment analysis, and image classification.

- **Unsupervised Learning**: Unsupervised learning involves training a model on data without labeled responses. The goal is to let the model explore the data and find its own structure within. Common unsupervised learning tasks include clustering, dimensionality reduction, and association mining.

- **Reinforcement Learning**: This type of learning uses a system of rewards and penalties to compel the computer to solve a problem by itself. Typically, we cannot provide an accurate expected output to every state in the input space because of the complexity of the task. Reinforcement learning algorithms learn to react to an environment on their own by trying to maximize the long-term rewards gained in the system. Applications include robotics, gaming, and navigation.

In this thesis, we concentrate on developing models to explore the characteristics and patterns inherent in Rubik's cube states to facilitate heuristic search. Specifically, we utilize supervised learning to assign labels to specific states of the Rubik's cube, and reinforcement learning, where the cube is treated as a system driven by a framework of rewards and penalties. These methodologies will be examined in greater detail in Chapter 4.

2.4.2 Deep Supervised Learning

Deep learning is a specialized subset of machine learning that utilizes deep neural networks (DNN) that emulate the structure and function of the human brain. Essentially, it is a machine learning technique that learns features and tasks directly from data through a "neural network". These networks are composed of layers of interconnected nodes or "neurons" shown in Fig. 2.8, each capable of performing simple linear transformation.

Architecture of Deep Neural Networks

Deep neural networks typically consist of one input layer, one output layer, and at least one hidden layer (denoted as m hidden layers where $m \geq 1$). These networks process data by recognizing patterns through training, ultimately making predictions based on new inputs. The fundamental operation in these networks involves adjusting the parameter set (θ) associated with each neuron, which is refined based on previous outputs and new data to enhance the accuracy of predictions.

The configuration of hidden layers (m) and the number of neurons within each layer critically influence the model's performance. Increasing the number of layers

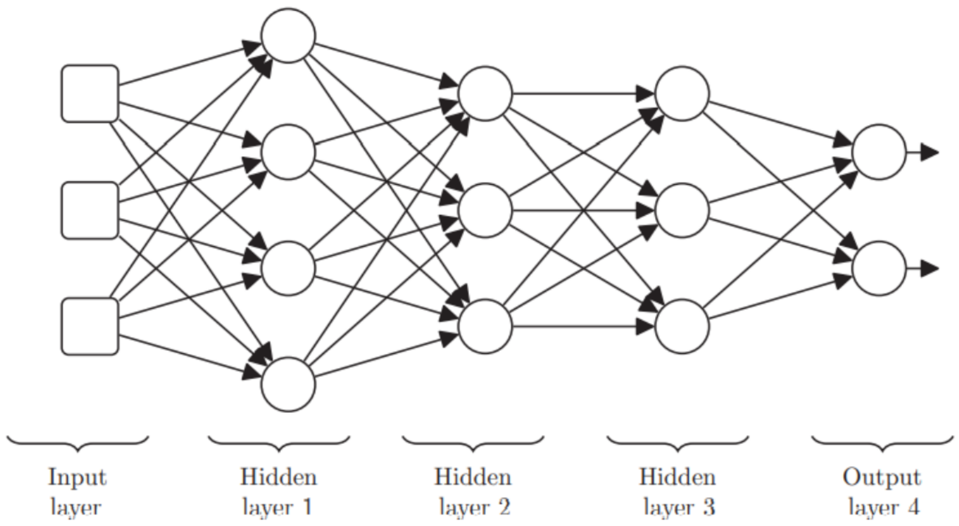


Figure 2.8: Neural Network Architecture (Figure From [Shr22])

allows the network to capture more complex patterns, but it also increases the computational demand and the risk of overfitting. The number of neurons per layer determines the network's capacity, impacting both computational requirements and the ability to generalize. Additionally, the connectivity pattern among neurons across different layers affects the network's training dynamics and performance.

This thesis will focus on fully connected feedforward neural networks among the three primary types of architectures, which also include recurrent neural networks and convolutional neural networks. In fully connected feedforward networks, every neuron in a layer is connected to each neuron in the subsequent layer without any backward connections, as illustrated in Fig. 2.8. This architecture offers robust approximation capabilities and flexible input size handling, although it is more susceptible to overfitting. Further discussion on this will follow in Chapter 4.

We will also delve into the training processes essential for developing both accurate and reliable neural network models using the supervised learning algorithm. This method comprises two key phases: forward and backward propagation. Forward propagation involves passing input data through the network to generate an output, which is then used to evaluate the model's performance. Backward propagation focuses on adjusting the model based on the errors identified during the forward phase, optimizing the parameters to improve outcomes in subsequent iterations.

Forward Propagation

The training starts with forward propagation. The purpose of forward propagation is to transmit the input data from the input layer to the output layer through various hidden layers. Consider a neural network with n neurons in the input layer $\{x_1, \ldots, x_n\}$, m hidden layers (where $m \geq 1$), and an output layer with k neuron.

For each layer, data representation is handled using vectors or matrices corresponding to the neuron contents. For instance, the input data vector for the input layer is represented as:

$$\boldsymbol{X} = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix}$$

This notation signifies the data transmitted to the input layer. Similarly, $\boldsymbol{H_i}$ represents the data within the i-th hidden layer, and $\boldsymbol{Y}$ represents the data obtained from the output layer. For clarity, we will consider $\boldsymbol{X}$ as a vector in this discussion to illustrate the process of propagation through the network layers.

The process begins at the input layer, which is linked to the first hidden layer via connections that are weighted by a matrix referred to as *weights*. Let $\boldsymbol{W_1}$ denote the weight matrix between the input layer and the first hidden layer. Suppose that the first hidden layer has t neurons. Then, the dimension of $\boldsymbol{W_1}$ should be $n \times t$ which ensures that the output dimension aligns with the number of neurons in the first hidden layer ($\boldsymbol{H_1}$).

Each $w_{i,j} \in \boldsymbol{W_1}$ represents the weight between the i-th neuron in the input layer and the j-th neuron in the first hidden layer. Specifically, The relationship between the input data $\boldsymbol{X}$ and the first hidden layer $\boldsymbol{H}$ can be mathematically represented as follows:

$$\begin{aligned} \boldsymbol{H_1} &= \boldsymbol{X_1} \times \boldsymbol{W_1} \\ &= \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix} \times \begin{bmatrix} w_{1,1} & \dots & w_{1,t} \\ w_{2,1} & \dots & w_{2,t} \\ & \dots & \\ w_{n,1} & \dots & w_{n,t} \end{bmatrix} \\ &= \begin{bmatrix} h_1 & h_2 & \dots & h_t \end{bmatrix} \end{aligned}$$

In addition to the weight matrix, each hidden layer is associated with a bias vector $\boldsymbol{B_i}$, which adjusts the linear combination of inputs and weights to help the network model more complex functions. With the addition of the bias vector, the data in the first hidden layer becomes

$$\boldsymbol{H_1} = \boldsymbol{X_1} \times \boldsymbol{W_1} + \boldsymbol{B_1}$$

where $\boldsymbol{B_1}$ represents the bias vector for the first hidden layer. For each neuron output h_j in $\boldsymbol{H_1}$, it can be explicitly calculated as:

$$h_j = \sum_{i=1}^{n} x_i \cdot w_{i,j} + b_j$$

An example can be seen in Fig. 2.9 with two neurons in the input layer and three neurons in the first hidden layer to visualize the calculation.

This resulting hidden layer vector $\boldsymbol{H_1}$ is then input to a non-linear activation function like sigmoid or ReLU, which determines which neurons fire to the next layer. Among the various activation functions, the **Rectified Linear Unit**, commonly abbreviated as ReLU, is widely adopted due to its simplicity and effectiveness. The ReLU function is defined as:

$$R(x) = \max(0, x)$$

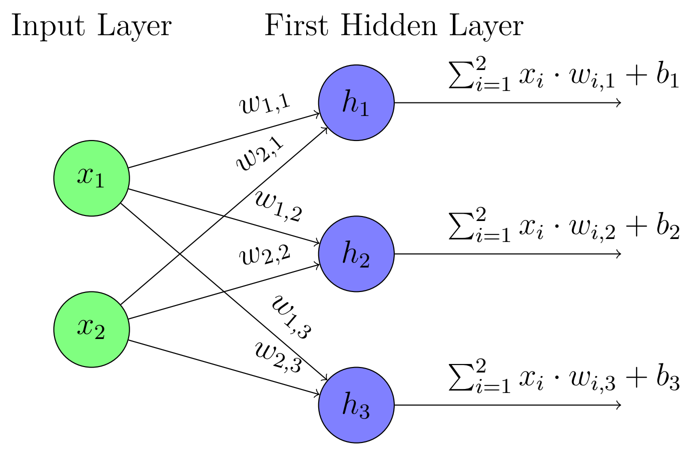

Input Layer
First Hidden Layer
x_1
x_2
h_1
h_2
h_3
$w_{1,1}$
$w_{2,1}$
$w_{1,2}$
$w_{2,2}$
$w_{1,3}$
$w_{2,3}$
$\sum_{i=1}^{2} x_i \cdot w_{i,1} + b_1$
$\sum_{i=1}^{2} x_i \cdot w_{i,2} + b_2$
$\sum_{i=1}^{2} x_i \cdot w_{i,3} + b_3$

Figure 2.9: Neural Network with Labeled Input and First Hidden Layers

In the context of neural networks, where we deal with vectors and matrices rather than just individual numbers, the ReLU function is applied element-wise. When applied to a matrix $\boldsymbol{X}$, the function is denoted as $A(\boldsymbol{X})$, where $\boldsymbol{X}$ is the input matrix. The activation function R operates on each element $x_{i,j}$ of the matrix for $A(\boldsymbol{X})$ which can be formally expressed as:

$$A(\boldsymbol{X}) = \begin{bmatrix} R(x_{1,1}) & \dots & R(x_{1,m}) \\ & \dots & \\ R(x_{n,1}) & \dots & R(x_{n,m}) \end{bmatrix} = \begin{bmatrix} \max(0, x_{1,1}) & \dots & \max(0, x_{1,m}) \\ & \dots & \\ \max(0, x_{n,1}) & \dots & \max(0, x_{n,m}) \end{bmatrix}$$

ReLU will be utilized throughout this thesis to introduce non-linearity after each layer, enabling the network to capture complex patterns in Rubik's Cube.

The necessity for non-linear activation functions like ReLU becomes apparent when considering the limitations of linear activation functions. Specifically, if a linear function is used across a multi-layered network, the collective effect of these layers becomes mathematically equivalent to that of a single-layer perceptron, no matter how many layers are stacked.

After the activation function, the activated output for the first hidden layer can be represented as Equation 2.4.4.

$$\hat{\boldsymbol{H}}_{\boldsymbol{1}} = A(\boldsymbol{X_1} \times \boldsymbol{W_1} + \boldsymbol{B_1}) \tag{2.4.4}$$

Continuing with the example shown in Fig. 2.9, the activated output of the first hidden layer is further converted with the activate function applied shown in Fig. 2.10.

The process of weight and bias application followed by activation continues layer by layer. For each two hidden layers, say $(i-1)$-th hidden layer and i-th hidden layer where $i \leq m$. Then the output of i-th hidden layer is

$$\hat{\boldsymbol{H}}_{\boldsymbol{i}} = A(\boldsymbol{H_{i-1}} \times \boldsymbol{W_i} + \boldsymbol{B_i})$$

With deeper hidden layers, the network can represent more complex inputs since each layer performs a transformation of the data passed through it. The final output

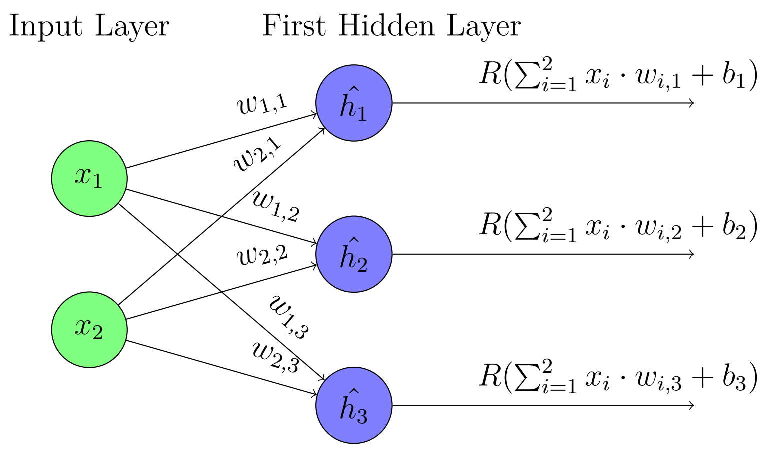

Input Layer
First Hidden Layer
x_1
x_2
$\hat{h}_1$
$\hat{h}_2$
$\hat{h}_3$
$w_{1,1}$
$w_{2,1}$
$w_{1,2}$
$w_{2,2}$
$w_{1,3}$
$w_{2,3}$
$R(\sum_{i=1}^{2} x_i \cdot w_{i,1} + b_1)$
$R(\sum_{i=1}^{2} x_i \cdot w_{i,2} + b_2)$
$R(\sum_{i=1}^{2} x_i \cdot w_{i,3} + b_3)$

Figure 2.10: Neural Network with ReLU

is derived using similar propagation from the last hidden layer to the output layer:

$$\hat{\boldsymbol{Y}} = \boldsymbol{H_m} \times \boldsymbol{W_Y}$$

The purpose of the deep learning model is to iteratively adjust the weight matrices $\boldsymbol{W_i}$ and bias vectors $\boldsymbol{B_i}$ in each hidden layer to optimize performance, typically through backpropagation which we will explore in the next section.

Backward propagation

Backward propagation is essential for learning in neural networks by adjusting the model's parameters based on the output error. The information is sent back to the hidden layers to adjust the parameter sets, $\boldsymbol{W_i}$ and $\boldsymbol{B_i}$, to minimize the loss. This process begins by calculating the loss, L, which measures the deviation between the predicted output $\hat{\boldsymbol{Y}}$ and the true output $\boldsymbol{Y}$ (since we are in the supervised learning framework), as described in Section 2.4.1. The loss is calculated from:

$$L = L(\boldsymbol{Y}, \hat{\boldsymbol{Y}})$$

To minimize L, we compute the gradient of L with respect to the predicted outputs $\hat{\boldsymbol{Y}}$, employing the partial derivative:

$$\nabla_{\hat{\boldsymbol{Y}}}(L) = \frac{\partial L}{\partial \hat{\boldsymbol{Y}}}$$

Note that $\hat{\boldsymbol{Y}} = \boldsymbol{H_m} \times \boldsymbol{W_Y}$, then we can apply the chain rule to compute the gradient of L with respect to $\boldsymbol{W_Y}$ such that

$$\nabla(L)_{\boldsymbol{W_Y}} = \frac{\partial L}{\partial \hat{\boldsymbol{Y}}} \cdot \frac{\partial \hat{\boldsymbol{Y}}}{\partial \boldsymbol{W_Y}}$$

Since the predicted output $\hat{\boldsymbol{Y}}$ is computed with a chain of functions based on the hidden layers. Mathematically, we have that

$$\hat{\boldsymbol{Y}} = (A \cdots (A(A(\boldsymbol{X_1} \times \boldsymbol{W_1} + \boldsymbol{B_1}) \times \boldsymbol{W_2} + \boldsymbol{B_2}) \cdots + \boldsymbol{B_{m-1}}) \times \boldsymbol{W_Y}$$

We apply the chain rule to compute the gradients of L with respect to each layer's parameters $\boldsymbol{W_i}$ and $\boldsymbol{B_i}$.

The optimizer then updates these parameters using the computed gradients. We utilize optimization algorithms like Gradient Descent ($\nabla = 0$), where the Adaptive Moment Estimation (*Adam*) is commonly applied, facilitating efficient and effective parameter adjustments. We will use *Adam* in this thesis throughout.

When the parameter set is optimized and adjusted using the optimizer based on the gradients and learning rate, we can start the next iteration from the forward propagate again. This process is repeated for a predefined number of iterations or until convergence.

Batch Processing for Training

Modern computing heavily relies on Graphics Processing Units (GPUs) for accelerating deep learning algorithms. GPUs, with their hundreds or thousands of small cores, are adept at handling matrix and vector operations concurrently. This capability is crucial for batch processing, where multiple data instances are processed simultaneously, greatly enhancing the efficiency of training and search processes in deep learning models.

GPUs excel in parallel processing, which is foundational to batch processing. The architecture of DNN primarily involves matrix manipulation and linear transformations, making them inherently suitable for batch processing.

Consider the scenario where we input three states into a DNN concurrently. Suppose each input state has a dimension of $1 \times x$. If we input three states s_1, s_2, s_3 into DNN at the same time, we are creating a matrix containing s_1, s_2, and s_3 simultaneously, we effectively convert three individual vectors into a single matrix:

$$\boldsymbol{S} = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,x} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,x} \\ x_{3,1} & x_{3,2} & \cdots & x_{3,x} \end{bmatrix}$$

The dimension for s is $3 \times x$. The weight matrix of the first hidden layer, $\boldsymbol{W_1}$, has a dimension of $x \times h_1$ given h_1 neurons in that layer. With batch input, the output of this hidden layer, $\boldsymbol{H_1}$, will be:

$$\boldsymbol{H_1} = \boldsymbol{S} \times \boldsymbol{W_1} + \boldsymbol{B_1}$$

resulting in dimensions of $3 \times h_1$. As we proceed through subsequent layers, the final output layer will also produce a batched output, each row corresponding to the predictions for each input state in matrix $\boldsymbol{S}$.

This structural approach allows for processing a matrix of dimensions $n \times m$ to yield outputs of $n \times o$ where o represents the neuron count in the output layer. With GPU support and DNN architecture optimized for batch processing, the computation time remains consistently around 1 second for an input size of around the magnitude of 10^6, regardless of the batch size. This performance is substantially better compared to single processing, demonstrating the significant efficiency gains achievable through batch processing in deep learning frameworks. Therefore, during training, data need not be fed into the model one at a time; instead, it can be input as batches, saving significant computational time.

2.4.3 Deep Reinforcement Learning

Deep reinforcement learning is an important technique in this thesis as the final goal is that we want to construct a heuristic function to measure the minimal steps to solve any given valid cube states of the Rubik's cube. To do this, we need to first introduce the basic framework of deep reinforcement learning on how can we use machines to train such functions.

Framing the Problem using the Markov Decision Process

As explained in Section 2.4.1, reinforcement learning focuses on how agents ought to take actions in an environment to maximize some notion of cumulative reward. It's rooted in the idea that an agent learns from the consequences of its actions, rather than from being told explicitly what to do. A core mathematical framework used in reinforcement learning is the Markov Decision Process (MDP), which provides a formalization of decision-making situations where outcomes are partly random and partly under the control of a decision-maker.

For this section, we will define the MDP of the Rubik's Cube directly as $M = (S, A, P, R)$ where each component is defined as

- State Space (S): A set of states representing the environment in which the agent operates. Each state encapsulates all the information necessary for the agent to make a decision at that point. In our case, the state space is the valid cube space containing all valid states of the Rubik's cube. The size of S is 4.3×10^{19}.

- Action Space (A): For each state, there is a set of possible actions the agent can take. Actions influence state transitions. As for the Rubik's cube, the action space A is just the 12 allowed moves from Definition 2.1.2. So, $A = \Psi$.

- Transition Function (P): This is a function that predicts the next state, given the current state and the action taken by the agent. It represents the dynamics of the environment and is often expressed as a probability distribution over the next states. In our case, the Rubik's cube state is deterministic which means that the next state is certain given the current state and the action taken. For example, if a state s' is the resulting state that applies `U` to a cube state s, then $P(s' \mid s, a) = 1$ since it cannot result in other states with the deterministic nature of cube moves.

- Reward Function (R): A function that assigns a reward (or penalty) to each transition between states. It reflects the goal of the task, guiding the agent by providing feedback on the desirability of outcomes. As for the environment of the Rubik's Cube, the immediate reward of transition from s to s' via an action a is deterministic as well and it is defined by:

$$R(s', s, a) = \begin{cases} 0 & \text{if } s' \text{ is } 0 \\ -1 & \text{Otherwise} \end{cases}$$

We use a four-tuple $M = (S, A, P, R)$ to denote the MDP for the Rubik's Cube environment which can be visualized as Fig. 2.11. We will explore how to solve it in the next section.

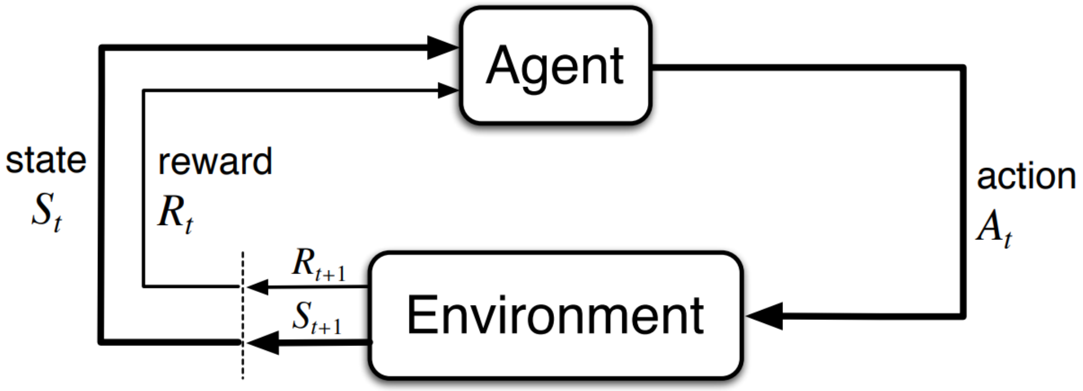


Figure 2.11: The agent–environment interaction in a Markov decision process from [SB18]

Goal of RL Regarding Value Function

The primary goal of reinforcement learning (RL) is to find an optimal policy that effectively addresses the Markov Decision Process (MDP) introduced earlier.

We define a **policy**, denoted by π, as a set of rules used by the agent to decide which actions to take in response to any given state of the environment. Since the environment in this context is deterministic as the actions are real cube moves, we represent the policy π as a function mapping states to actions:

$$\pi : S \to A$$

To determine the most effective policy, it is essential to assess the expected returns from each state, which reflect the total rewards an agent can accumulate over time. Rather than focusing solely on short-term or long-term rewards, we examine the value function of a policy π, denoted V_π. Typically, a discount factor γ would be used in V_π to prioritize immediate rewards over future ones. However, to treat all rewards equally in our environment since they have equal importance, we set $\gamma = 1$ and omit it from the notation. The value of a state s under policy π, $V_\pi(s)$, is the expected sum of rewards starting from state s and following policy π:

$$V_\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} R_k \mid s_0 = s \right]$$

where R_k represents the rewards received at time k.

We further define $V^*(s)$ as the optimal value function, representing the highest value attainable at state s under the best possible policy, denoted as π^*:

$$V^*(s) = \max_{\pi} \mathbb{E}_{\pi} \left[\sum_{k=0}^{\infty} R_k \mid s_0 = s \right]$$

This leads to the Bellman Equation, which maintains optimality by considering the value of subsequent states. [SB18] Notice that V^* is the sum of all future rewards. We can break it up into the current reward at the time step now (s_0) plus the rewards from the next state s' to infinity. We have that

$$V^*(s) = \max_{\pi} \mathbb{E}_{\pi} \left[R_0 + \sum_{k=1}^{\infty} R_k \mid s_1 = s' \right]$$

This simplifies to:

$$V^*(s) = \max_{\pi} \mathbb{E}_{\pi} \left[R_0 + V^*(s') \right]$$

It is known as the Bellman equation. [SB18] It has the power that if we know what the best next state is, all we have to do now is to perform the action that can lead the current state to that state. It is a recursive expression that is essentially breaking the problem into sub-problems.

From V^*, the optimal policy π^* can always be derived as it maximizes the expected reward:

$$\pi^* = \operatorname*{argmax}_{\pi} \mathbb{E}_{\pi} \left[R_0 + V^*(s') \right]$$

Thus, the problem reduces to solving for the value function V^*. We will explore the method of resolving V^* to derive π^* in Chapter 4, utilizing a combination of dynamic programming and deep neural networks.

Chapter 3

The Rubik's Cube and Group Theory

This chapter delves into the application of group theory to refine our understanding of the Rubik's Cube state space, specifically targeting the recognition of invalid states and the substantiation of the first cubology theorem proposed by [Ban82]. Leveraging Rubik's group, we will also introduce an algorithm designed to validate cube states, effectively circumventing the generation of redundant states in the course of our search.

However, it is crucial to clarify what "valid" means in this context and how we define the valid states of the Rubik's Cube. To do so, we introduce two definitions to distinguish between "possible states" and "valid states" of the cube.

Definition 3.0.1. Possible States: A possible state refers to any conceivable configuration of the Rubik's Cube that can be achieved by manipulating the individual cubies, except for the fixed center pieces. Imagine disassembling the Rubik's Cube and then randomly reassembling it; any configuration you could achieve in this way represents a possible state.

This definition is distinct from scenarios where stickers are removed and randomly replaced, which would affect all 54 facets, represented mathematically as S_{54}. While all possible states resulting from the manipulation of cubies are elements of S_{54}, not every permutation within S_{54} corresponds to a physically achievable state of the cube, due to the constraints imposed by its mechanical design.

Definition 3.0.2. Valid States: Conversely, a valid state of the Rubik's Cube is a configuration that can be reached by performing a series of allowed moves in Ψ, as defined in Definition 2.1.2, starting from the solved state. In other words, a valid state can be achieved and returned to the solved configuration through legal moves alone, without the need to disassemble the cube.

For instance, the configuration that merely swaps two edge pieces, illustrated in Fig. 3.1, is considered as a possible state of the Rubik's Cube. But as we will prove in Section 3.3, this configuration cannot be solved with combinations of the allowed moves. So, it is invalid but possible.

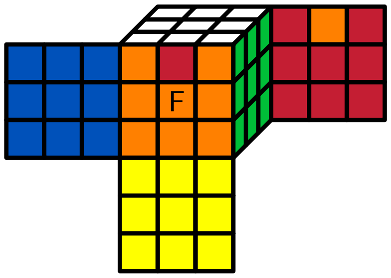


Figure 3.1: Invalid state of the Rubik's cube

First, we will develop a notation system to describe the state of the Rubik's Cube. This system will allow us to clearly and systematically represent the permutations and orientations of all the cubies, which is crucial for analyzing, solving, and discussing the cube's configurations.

3.1 Representation of the Rubik's Cube States

On a cubie level, the Rubik's Cube consists of center cubies, corner cubies, and edge cubies. Therefore, the state of a Rubik's Cube will be parameterized by four criteria: corner cubie permutation, corner cubie orientation, edge cubie permutation, and edge cubie orientation. We will examine each one in this section.

3.1.1 Corner Cubie Encoding

We define the ***permutation*** of corner cubies, denoted as α, to represent the current corner cubies' positions in relation to their original positions in the solved state.

A numerical identification is assigned to the slots designated for corner cubies, referenced according to their relative positions to the up and down faces of the cube. Additionally, we employ a naming convention for both the slots and the corner cubies that follows a clockwise sequence relative to the cube's faces. For instance, as depicted in Fig. 3.2, the corner cubie situated at the intersection of the Upper, Right, and Front faces is encoded as `URF`.

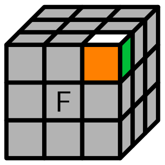


Figure 3.2: The illustration of corner cubie/slot `URF`.

To make the permutation meaningful and easier to identify, we assign specific numbers to the corner cubie slots. The slot numbering is visualized in Fig. 3.3 and is assigned as follows:

Slot 1: The URF corner of the cube
Slot 2: The UFL corner of the cube
Slot 3: The ULB corner of the cube
Slot 4: The UBR corner of the cube
Slot 5: The DFR corner of the cube
Slot 6: The DLF corner of the cube
Slot 7: The DBL corner of the cube
Slot 8: The DRB corner of the cube

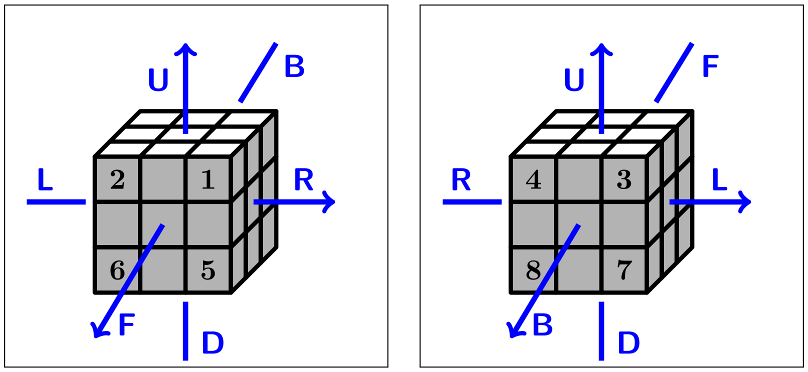


Figure 3.3: The visualization of the corner cubie slots.

With the slots defined, it becomes essential to identify which corner cubie occupies each slot, leading us to the concept of a corner permutation. To facilitate this, we assign a unique code to each corner cubie based on its position in the solved state:

C_1 to denote the URF corner cubie in the solved state
C_2 to denote the UFL corner cubie in the solved state
C_3 to denote the ULB corner cubie in the solved state
C_4 to denote the UBR corner cubie in the solved state
C_5 to denote the DFR corner cubie in the solved state
C_6 to denote the DLF corner cubie in the solved state
C_7 to denote the DBL corner cubie in the solved state
C_8 to denote the DRB corner cubie in the solved state

The permutation of corner cubies is then represented by an element in S_8, denoted as α. In α, the i-th index holding a cubie C_j (where $j \in \{1, \ldots, 8\}$) indicates that the corner cubie currently in the i-th slot corresponds to C_j in the solved state. Note that α represents the positions of corner cubies, differing from traditional permutations in S_8 which describe abstract arrangements of eight elements.

For example, the initial solved state permutation α is $(C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8)$ which is $1 \in S_8$ where the 1 here represents the identity element in S_8. After executing a `R` move, the permutation of α changes to $(C_5, C_2, C_3, C_1, C_8, C_6, C_7, C_4)$ which is equal to $(1\ 5\ 8\ 4)$ followed the procedure from Remark 2.2.9, reflecting the new positions of the corner cubies due to the move.

The use of C_j to describe each cubie's placement provides a clear and precise way to articulate the permutation of corner cubies as an element of the symmetric group S_8. We will use this C_j to denote α which is the same as writing it as a permutation in S_8.

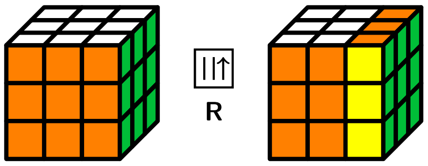


Figure 3.4: Example of executing `R` to a solved cube.

Having established permutation, let's move to ***orientation***. In this context, orientation refers to the arrangement of individual corner cubies rather than the entire cube. Every corner cubie has three potential orientations, which we label as 0, 1, and 2.

Here's how to interpret these labels: Starting with the cube in its solved configuration, assign the numbers 0, 1, and 2 to each face of a corner cubie. The rule is straightforward: Faces pointing upwards or downwards get a 0. Surrounding faces are assigned 1 and 2 in an anticlockwise sequence. As an illustration, if you're holding a solved cube with the orange side front and white side up, the URF corner's white face gets a 0, its orange face gets a 1, and the green face gets a 2. Refer to the subsequent illustration for clarity like we see in Fig. 3.5.

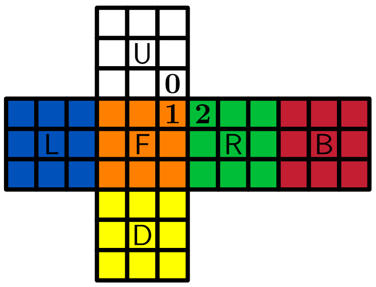


Figure 3.5: Orientation Encoding for `URF` corner cubie.

To accurately understand the orientation of corner cubies on the Rubik's Cube, it's essential to recognize how each face's original position in the solved state informs the orientation throughout any sequence of moves. This principle ensures that the assignment of numbers to faces, which indicate their position in the solved state, remains consistent even as the cube is scrambled or solved. Given that each corner cubie displays three distinct colors, identifying the orientation of one face automatically determines the orientation of the other two faces in an anticlockwise direction.

For instance, the orientation of the `URF` cubie can be uniquely determined by this method. It is incorrect for the green face of this cubie to be positioned on the right side (as would be suggested in an incorrect orientation 2 in Fig. 3.6) if we follow the encoded system. If the white face is on the top, then the correct orientation is as depicted in case 1, where the orange facet is positioned anticlockwise to the white facet.

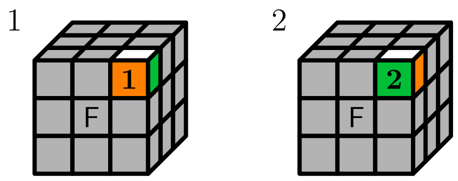


Figure 3.6: Correct (Case 1) and incorrect (Case 2) orientations of the `URF` corner cubie

To depict the orientation of each cubie, we employ an 8-tuple vector, symbolized as γ which is in $\mathbb{Z}_3^8$. Each index i in γ records the number facing **up** or **down** for the cubie in the corresponding i-th slot. In the solved state, all corners have the up or down face pointing up or down, thus:

$$\gamma = (0, 0, 0, 0, 0, 0, 0, 0).$$

Executing a U move won't alter γ as all white and yellow faces persistently point up or down. Conversely, an R move impacts orientation as seen in Fig. 3.7.

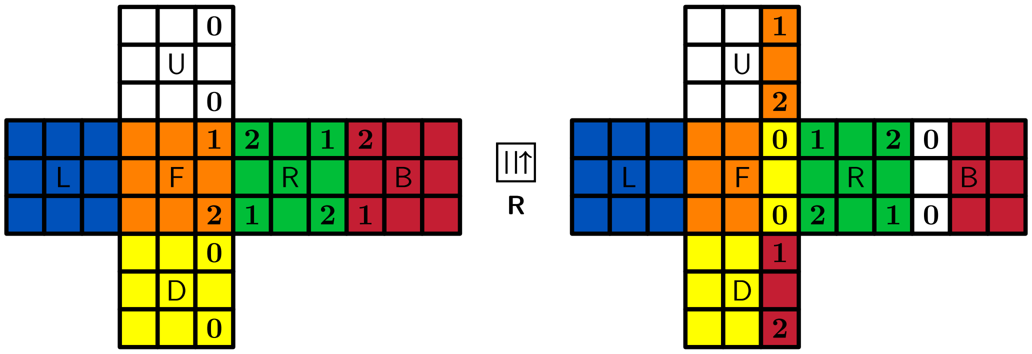


Figure 3.7: The rotations `R` on a solved cube.

From Fig. 3.7, the corner cubie in slot 1 (DFR) has its orange face upwards. This face was originally assigned the number 2. Therefore, the orientation of DFR in slot 1 is 2, leading to the 0th position in γ being 2. Using this logic, we can deduce the entire orientation vector:

$$\gamma = (2, 0, 0, 1, 1, 0, 0, 2).$$

Consequently, the state of the cube's corner cubies after an `R` move can be represented as a combination of two 8-tuple vectors: the permutation, α, and the orientations, γ where

$$\alpha = (C_5, C_2, C_3, C_1, C_8, C_6, C_7, C_4)$$

and

$$\gamma = (2, 0, 0, 1, 1, 0, 0, 2).$$

Now, we have the tools to encode corner cubies and write the configuration of the corner cubies. Together, α and γ unambiguously define the configuration of the corner cubies by specifying both which cubie is in each slot (α) and how each cubie is oriented within its slot (γ).

3.1.2 Edge Cubie Encoding

In a manner analogous to the corner cubies, we shall now establish a formal representation of the state of the edge cubies of a Rubik's Cube. First, we enumerate the edge positions on the cube to assign each with a unique slot number:

Slot 1: The UR edge of the cube
Slot 2: The UF edge of the cube
Slot 3: The UL edge of the cube
Slot 4: The UB edge of the cube
Slot 5: The DR edge of the cube
Slot 6: The DF edge of the cube
Slot 7: The DL edge of the cube
Slot 8: The DB edge of the cube
Slot 9: The FR edge of the cube
Slot 10: The FL edge of the cube
Slot 11: The BL edge of the cube
Slot 12: The BR edge of the cube

Similarly, we assign a unique code to each edge cubie based on its position in the solved state:

E_1 to denote the UR edge in the solved state
E_2 to denote the UF edge in the solved state
E_3 to denote the UL edge in the solved state
E_4 to denote the UB edge in the solved state
E_5 to denote the DR edge in the solved state
E_6 to denote the DF edge in the solved state
E_7 to denote the DL edge in the solved state
E_8 to denote the DB edge in the solved state
E_9 to denote the FR edge in the solved state
E_{10} to denote the FL edge in the solved state
E_{11} to denote the BL edge in the solved state
E_{12} to denote the BR edge in the solved state

We encapsulate the ***permutation*** of the edge cubies with a permutation in S_{12}, denoted by β. The i-th element of β, noted as β_i, signifies which edge cubie E_j resides in the i-th slot in the cube's current configuration based on the starting assignment of edges. Similarly, the solved cube will have an edge permutation of

$$\beta = (E_1, E_2, E_3, E_4, E_5, E_6, E_7, E_8, E_9, E_{10}, E_{11}, E_{12}) = 1 \in S_{12}$$

Each edge cubie can be oriented in two ways, which we numerically define as 0 and 1. To illustrate, in the cube's solved state, any face of an edge cubie that is oriented towards the **up** (`U`), **down** (`D`), **front** (`F`), or **back** (`B`) side is labeled as 0. Conversely, if it faces the **left** (`L`) or **right** (`R`) side, it is labeled as 1.

For instance, holding a solved cube with the orange face front and the white face up, the white face of the UR edge cubie is labeled as 0, and its adjacent green face as 1 which is shown in Fig. 3.8.

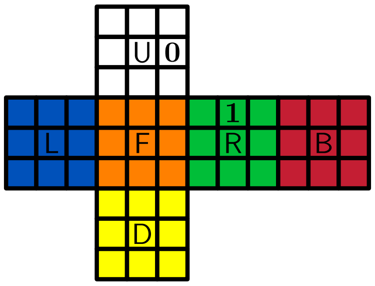


Figure 3.8: Orientation Encoding for `UR` corner cubie

The ***orientation*** for each edge is recorded in a 12-tuple vector in $\mathbb{Z}_2^{12}$ as δ. Note that each δ_i also uniquely determines the orientation of the cubie at i-th slot with the same reason as the corner cubie orientation.

Let's continue with the case of the `R` move, we can now write out the representation for edge cubies. In the solved state, $\beta = 1$ and $\delta = \mathbf{0}$. If we do an `R` move shown in the Fig. 3.9, then β changes to $(E_9, E_2, E_3, E_4, E_{12}, E_6, E_7, E_8, E_5, E_{10}, E_{11}, E_1)$ which is equal to $(1\ 9\ 5\ 12) \in S_{12}$. However, in this case, δ doesn't change based on the figure.

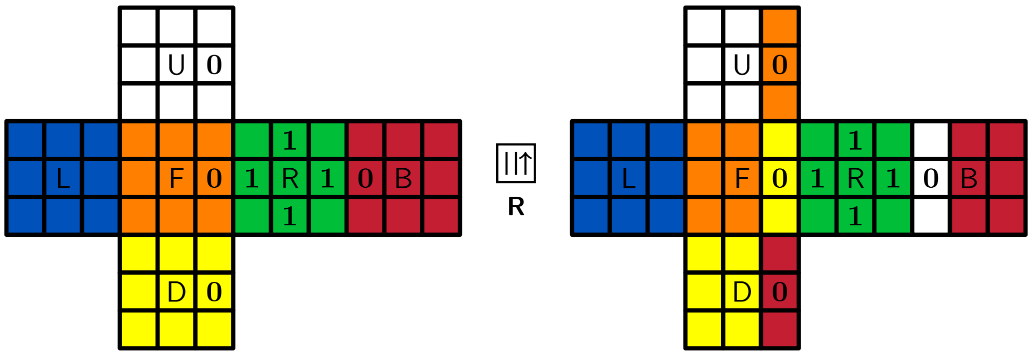


Figure 3.9: The rotations **R** on a solved cube.

3.1.3 Comprehensive Example

To illustrate the practical application of our notation system, let's examine a cube scramble. Begin with the cube-oriented with the orange face fronting and the white face up. Then, apply the sequence `U R F` which will result in the state as shown in Fig. 3.10.

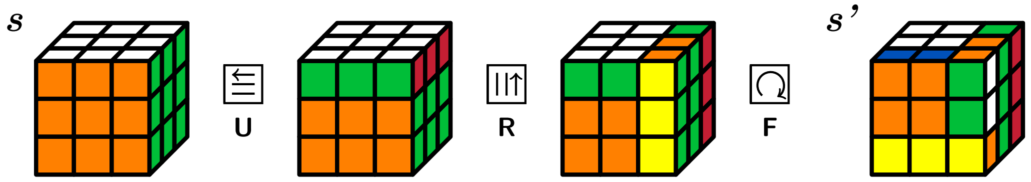


Figure 3.10: The rotations **U**, **R**, **F** on a solved cube.

The resulting cube state, s', can be represented as a 4-tuple vector $(\alpha, \beta, \gamma, \delta)$. For this scramble, the specific vectors are:

$$\alpha = (C_1, C_6, C_2, C_4, C_5, C_8, C_7, C_3)$$
$$\beta = (E_9, E_{10}, E_2, E_3, E_{12}, E_5, E_7, E_8, E_1, E_6, E_{11}, E_4)$$
$$\gamma = (1, 2, 0, 1, 1, 2, 0, 2)$$
$$\delta = (0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0)$$

Note here each α_i paired with γ_i uniquely determines the specific corner cubie in the i-th slot and its orientation among its 3 possible states. For example, $\alpha_2 = C_6$ means that the `DLF` corner cubie is positioned in the `UFL` slot. And $\gamma_2 = 2$ which indicates that the facet originally marked with 2 is facing upwards. Thus, the combination of α_2 and γ_2 allows us to deduce the precise cubie and its orientation in the `UFL` slot. Fig. 3.11 illustrates the positioning and orientation of the corner cubie at the `UFL` slot.

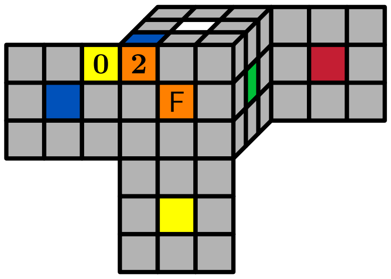


Figure 3.11: Corner cubie at the `UFL` slot

Through decoding permutations and orientations repeatedly with $\alpha, \beta, \gamma, \delta$, we can reconstruct the entire state of the Rubik's Cube. This 4-tuple vector uniquely represents any cube state, facilitating the detailed analysis and solving of the puzzle.

Definition 3.1.1. We define such representation of 4-tuple as the ***cubie-level cube representation*** where a state of the cube s can be represented as follows:

$$s = (\alpha, \beta, \gamma, \delta)$$

where

- α represents the permutation of the 8 corner cubies.
- β represents the permutations of the 12 edge cubies.
- γ represents the orientation of the corner cubies.
- δ represents the orientation of the edge cubies.

3.2 The Rubik's Cube Group

Using the cubie-level representation of Rubik's Cube states established in the previous section, we define the set of all **possible** states of the Rubik's Cube, denoted as S. Note that the possible states of the Rubik's cube can be represented using the cubie-level cube representation without any restrictions on the coordinates, i.e., α can be any permutation in S_8 and so on. And the set of all **valid** states as VS. Note that all valid states are possible states, so $VS \subset S$.

Defined in Definition 3.0.1, elements in S are any combinations of the 4-tuple and we know precisely the size of each tuple, so we can calculate the order of S as:

$$\begin{aligned} |S| &= |S_8| \cdot |S_{12}| \cdot |\mathbb{Z}_3^8| \cdot |\mathbb{Z}_2^{12}| \\ &= 8! \cdot 12! \cdot 3^8 \cdot 2^{12} \\ &= 5.19 \times 10^{20} \end{aligned}$$

In this chapter, we will explore the properties of valid states of the Rubik's Cube and determine the total number of valid states, $|VS|$.

3.2.1 Group Structure of the Rubik's Cube Group

To further understand the distinctions between valid and invalid states of a Rubik's Cube, we need to explore the concept of the Rubik's Cube group, denoted by G. This group comprises every possible move and combination of moves achievable on the cube, represented by the effects of any sequence of face rotations.

The operation for G is the composition of cube moves. For any $m_a, m_b \in G$, then $m_a \circ m_b$ means that apply m_b first and then apply m_a secondly. However, with the convention of writing the moves, we write $m_b\ m_a$ to denote $m_a \circ m_b$. Let Ω be the set of basic moves {`U, D, F, B, R, L, e`}, then a cube move is a sequence of elements of Ω.

If m, n are cube moves, we define $m \circ n$ as the concatenation of the string. For example, let m_L be applying the left layer with clockwise rotation and $m_{U'}$ be applying the up layer with anticlockwise rotation, and let $m \in G$ be "applying left layer with clockwise rotation, followed by an up layer anticlockwise rotation", then we write $m =$ `L U'` $= m_{U'} \circ m_L$.

Importantly, the moves of the Rubik's Cube inherently define its action, as each move in Ω has a direct impact on the cube's configuration. Thus, we define G to be the group of cube moves factored by the equivalence relations. For any $m_a, m_b \in G$ are in the same equivalence class if they do the same thing to the cube. For example, $m =$ `L L L L` is the same as e which are both doing nothing to the cube. We denote such m_a and m_b as $m_a \sim m_b$.

Hence, G is the group generated by Ω, factored by the equivalence relation $\sim$. Intuitively, the Rubik's Cube group G is the set of all distinct cube configurations reachable by sequences of moves from Ω, considering simplifications where redundant sequences are equated to simpler or null moves. As a result of this structure, there exists a one-to-one correspondence between the set of valid states, VS, and the group G. Each state in VS corresponds uniquely to a sequence of moves in G that transforms the solved cube to that state.

Previously, we observed how cube moves affect the cube's state through physical manipulation and empirical evidence, lacking a rigorous mathematical framework to formalize these observations. Each element of G can be correlated to permutations within S_8 and S_{12}, representing the permutations of the cube's corners and edges, respectively, along with vector modification over $\mathbb{Z}_3^8$ and $\mathbb{Z}_2^{12}$ for orientation adjustments. We formally define a cube move in G with Definition 3.2.1.

Definition 3.2.1. Let $m \in G$, then $m = (\sigma, \tau, \theta, \phi)$, where:

- $\sigma \in S_8$ and $\tau \in S_{12}$ correspond to the permutations of the cube's eight corners and twelve edges.
- θ and ϕ correspond to the vector modification over ${\mathbb{Z}_3}^8$ and ${\mathbb{Z}_2}^{12}$, reflecting the orientations of the cube's corners and edges done by m.

We will look at how m acts on the cube state s individually. It's crucial to differentiate between the permutations σ in the Rubik's group and the corner permutation α in the cube state representation, despite both being elements of S_8. In this context, α_i specifies which corner cubie is in the i-th slot, while σ_i delineates the intended swap or movement of corner cubies in the slots. For example, the move R implies a permutation $\sigma_R = (5\ 1\ 4\ 8)$, indicating the repositioning of corner cubies according to the move's effect—shifting the cubie from the 5th slot to the 1st, the 1st to the 4th, and so on.

Let's consider an additional $m =$ `R` move applied to the resulting cube state shown in Fig. 3.11. We'll describe how m acts on a state. As shown before, the initial corner permutation is

$$\alpha = (C_1, C_6, C_2, C_4, C_5, C_8, C_7, C_3)$$

applying $\sigma_R = (5\ 1\ 4\ 8)$ transforms α into

$$\sigma_R(\alpha) = \alpha' = (C_5, C_6, C_2, C_1, C_3, C_8, C_7, C_4)$$

presented in Fig. 3.12. This transformation exemplifies how permutations work to model the cube's transitions from one state to another.

The meaning of τ is analogous to σ, yet distinct from β. Just as σ represents the permutation of corner cubies, τ denotes the permutation of edge cubies. For

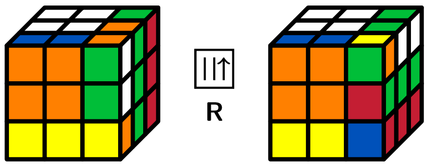


Figure 3.12: Executing R on a previous example

instance, $\tau_R = (1\ 12\ 5\ 9)$ illustrates the repositioning effect of a move—specifically, an R move—on edge cubies: the cubie in the 1st slot moves to the 12th slot, the cubie in the 12th slot moves to the 5th, and so on through the pattern specified by τ_R.

Given the impact of certain moves on the orientation of cubies, we employ the functions θ and ϕ to denote these orientation alterations. Specifically, θ represents the orientation shifts of corner cubies, while ϕ denotes the changes in the orientation of edge cubies.

The functions θ and ϕ consist of two components: the permutations σ and τ paired with the move sequence, and the vectors x and y that indicate the vector addition to the original orientation, respectively. For instance, for any orientation vector $\gamma \in \mathbb{Z}_3^8$ for corner cubies, executing a move sequence $m \in G$ is formalized by:

$$\theta_m(\gamma) = \sigma_m \circ ([x_m + \gamma]_3),$$

resulting in the updated orientation vector:

$$\gamma' = \theta_m(\gamma) = \sigma_m \circ ([x_m + \gamma]_3),$$

where $[n]_3$ denotes the modulo 3 addition, cycling the orientation within its three possible states.

Analogously, for the orientation vector $\delta \in \mathbb{Z}_2^{12}$ of edge cubies, the effect of a move sequence $m \in G$ is captured by:

$$\phi_m(\delta) = \tau_m \circ ([y_m + \delta]_2),$$

yielding the updated orientation:

$$\delta' = \phi_m = \tau_m \circ ([y_m + \delta]_2),$$

where $[n]_2$ is the addition modulo 2, reflecting the binary nature of edge orientations.

For instance, consider the effect of the R move. This move not only repositions certain cubies but also alters their orientation. Specifically, the corner cubie originally in the 1st slot is moved to the URB (Upper Right Back) position and undergoes an anticlockwise rotation, affecting its orientation. As a result, the orientation of the cubie initially at the 1st slot is increased by $1 \mod 3$, now occupying the 4th slot, to account for the anticlockwise turn. The "increase" in orientation is captured by x_{R1}, while the permutation is facilitated by σ_R. The cubie initially in the 4th slot moves and experiences a clockwise rotation, which effectively decreases its orientation value by $1 \mod 3$. Subsequently, it is shifted to the 8th slot. Although the rotation and permutation occur simultaneously, we separate these processes for simplicity. The meaning of y_R is similar to x_R.

To write it out, the `R` move's effect on corner cubie orientations, represented by θ_R, is given as

$$\theta_R = \sigma_R \circ x_R,$$

where

$$\sigma_R = (5\ 1\ 4\ 8) \quad \text{and} \quad x_R = ([1]_3, 0, 0, [-1]_3, [-1]_3, 0, 0, [1]_3).$$

Similarly, the effect on the edge cubies' orientations, represented by ϕ_R, is described as

$$\phi_R = \tau_R \circ y_R,$$

where

$$\tau_R = (1\ 12\ 5\ 9) \quad \text{and} \quad y_R = \mathbf{0} \in \mathbb{Z}_2^{12}.$$

It's noteworthy that the `R` move does not alter the orientation of any edge cubies, indicated by $y_R = 0$, meaning that the original orientation vector δ is effectively unchanged. Thus, ϕ_R solely facilitates the permutation defined by τ_R, without modifying the orientation values of the edge cubies.

Given that the permutations σ and τ are predefined for every move sequence, the real variables stored within θ and ϕ are x and y. Therefore, in subsequent analysis, keeping track of x and y—the vectors indicating the addition to the original orientation—becomes paramount. These vectors embody the critical adjustments to cubie orientations following a sequence of moves.

To summarize, let $m = $ `R`, then $m = (\sigma, \tau, \theta, \phi)$ acts s from Fig. 3.10 to form the resulting cube state s' where

$$s' = m(s) = (\sigma_m(\alpha), \tau_m(\beta), \theta_m(\gamma), \phi_m(\delta))$$

Summary Of Generator Moves

Table 3.1 contains all generators of G and their 4-tuple behaviors on any state $s \in S$.

Table 3.1: Generators of G and their effects on any state $s \in S$.

Move	σ	τ	x	y
U	(1 2 3 4)	(1 2 3 4)	0	0
D	(8 7 6 5)	(8 7 6 5)	0	0
R	(5 1 4 8)	(1 12 5 9)	$([1]_3, 0, 0, [-1]_3, [-1]_3, 0, 0, [1]_3)$	0
L	(3 2 6 7)	(3 10 7 11)	$(0, [-1]_3, [1]_3, 0, 0, [1]_3, [-1]_3, 0)$	0
F	(2 1 5 6)	(10 2 9 6)	$([-1]_3, [1]_3, 0, 0, [1]_3, [-1]_3, 0, 0)$	y_F
B	(8 4 3 7)	(12 4 11 8)	$(0, [-1]_3, [1]_3, 0, 0, [1]_3, [-1]_3)$	y_B

In the table, y_F and y_B are represented as follows:

$$y_F = (0, [1]_2, 0, 0, 0, [1]_2, 0, 0, [1]_2, [1]_2, 0, 0)$$
$$y_B = (0, 0, 0, [1]_2, 0, 0, 0, [1]_2, 0, 0, [1]_2, [1]_2)$$

For any two move sequences $m_1, m_2 \in G$, $m_1 \circ m_2 = (\sigma_1 \circ \sigma_2, \tau_1 \circ \sigma_2, \theta_1 \circ \theta_2, \phi_1 \circ \phi_2)$. This operation is well-defined as explained as follows:

- $\sigma_1 \circ \sigma_2$ means that we are first permuting by the rule of σ_2 and σ_1 which means the permutation of corner cubies doing m_2 then m_1. Same for $\tau_1 \circ \tau_2$.
- For $\theta_1 \circ \theta_2$ which is decomposed as $\sigma_1(x_1 + (\sigma_2(x_2 + \gamma)))$. It means we first do m_2 accounting for both the permutation and the orientation by θ_2 followed by θ_1. Note that the result from $\sigma_2(x_2 + \gamma)$ is still a vector in $\mathbb{Z}_3^8$ which can be further applied to the next operation. Same for $\phi_1 \circ \phi_2$.

The effect of the first move is applied before the second, corresponding to the real-world action of making one move on the Rubik's Cube followed by another.

3.2.2 The Set of All Valid Cube States

Now, we want to assess how G interacts with the set S since ultimately we are doing moves to the actual cube states. We have seen some of the examples before as the effect of the physical moves on the Rubik's Cube. In this section, we are going to formalize it using group action so that we have a framework for communicating complicated ideas.

The power of group theory really leverages when we enter the world of group action, a concept that allows us to apply the abstract properties of groups to more concrete mathematical objects and structures.

Definition 3.2.2. Let G be a group with the binary operation of $*$ and let X be any set. Then a group action is a map

$$\phi : G \times X \to X \qquad (g, x) \mapsto g * x$$

that satisfies:

1. $e * x = x$
2. $g * (h * x) = (gh) * x$

If G acts on a set X, then the orbit of $x \in X$ under G is the set $\{x \cdot g \mid g \in G\}$, denoted as $\mathrm{Orb}(x)$.

Consider the Rubik's Cube group G acts on the set of all possible states of the Rubik's cube S by Φ such that

$$\Phi : G \times X \to X$$

with the operation of

$$\Phi(m, s) : m(s) = (\sigma_m(\alpha_s), \tau_m(\beta_s), \theta_m(\gamma_s), \phi_m(\delta_s))$$

for some $m \in G$ and $s \in X$.

Let's check whether the map Φ is well-defined. Doing an "empty" move e to a state will not change the cube state, so $\Phi(e, s) = s$. As for the associativity of the operation. Let $g, h \in G$ and $s \in S$, then

$$\begin{aligned}
\Phi(g, \Phi(h, s)) &= \Phi(g, (\sigma_h(\alpha_s), \tau_h(\beta_s), \theta_h(\gamma_s), \phi_h(\delta_s))) \\
&= (\sigma_g(\sigma_h(\alpha_s)), \tau_g(\tau_h(\beta_s)), \theta_g(\theta_h(\gamma_s)), \phi_g(\phi_h(\delta_s))) \\
&= (\sigma_g(\sigma_h(\alpha_s)), \tau_g(\tau_h(\beta_s)), \theta_g(\theta_h(\gamma_s)), \phi_g(\phi_h(\delta_s))) \\
&= ((\sigma_g \circ \sigma_h)(\alpha_s)), (\tau_g \circ \tau_h)(\beta_s)), (\theta_g \circ \theta_h)(\gamma_s)), (\phi_g \circ \phi_h)(\delta_s))) \\
&= \Phi(g \circ h, s)
\end{aligned}$$

The verification steps confirm that Φ is a well-defined map, faithfully representing the effects of applying Rubik's Cube moves as group actions on the set of possible states. This demonstrates that Φ provides a mathematically sound model for the real-world manipulation of the Rubik's Cube.

The *orbit* of s_0 under the action Φ defined by the group G captures all configurations that can be reached from s_0 through the application of any sequence of moves in G from Definition 3.2.2. Formally, this is expressed as:

$$\mathrm{Orb}(s_0) = \{\Phi(g, s_0) \mid g \in G\} = \{g(s_0) \mid g \in G\},$$

which we denote as VS, the set of all valid configurations of the Rubik's Cube. Thus, $VS = \mathrm{Orb}(s_0)$.

The *stabilizer* of s_0 in G, denoted $\mathrm{Stab}(s_0)$, is the set of all elements in G that leave s_0 unchanged:

$$\mathrm{Stab}(s_0) = \{g \in G \mid g(s_0) = s_0\}.$$

Given the nature of the Rubik's Cube and the definition of G, where every element represents a distinct move sequence, the only element that does not alter s_0 is the identity element e. Therefore, $\mathrm{Stab}(s_0) = \{e\}$.

According to the orbit-stabilizer theorem [DF04], the size of the orbit of s_0 is given by the index of the stabilizer in G:

$$|\,\mathrm{Orb}(s_0)| = |VS| = \frac{|G|}{|\,\mathrm{Stab}(s_0)|}.$$

This relationship illustrates that each move sequence in G corresponds uniquely to one valid state of the Rubik's Cube, demonstrating that the total number of distinct, reachable configurations equals the order of G. Therefore, the orbit-stabilizer theorem not only confirms the validity of each configuration reached by moves in G but also underscores the comprehensive nature of G in encapsulating all valid states of the cube.

3.2.3 Prevention Algorithm on Reducing Replicated States

This section demonstrates how we can apply the Rubik's cube group G on practical tasks like expanding nodes in a search algorithm. In search algorithms applied to the

Rubik's Cube, it's essential to efficiently explore different states of the cube to find the solution. Initially, we start with the scrambled cube and apply all possible moves to it, evaluating how close each resulting state is to the solved state. As we progress, we select specific states based on the heuristic function for further expansion, meaning we apply additional moves to those states as discussed before. Section 2.3

However, this process can lead to redundant exploration. For instance, if we've already explored a state achieved by applying a sequence of moves, say "`U L U`", to the scrambled cube, and then we consider a move `U'` on this state, it takes us back to a previously explored state (achieved with "`U L`"), since `U'` undoes the last move `U`. Thus, applying `U'` immediately should be avoided as it results in an already explored state. To generalize it, if the last move is `X` where `X` $\in \Psi$, then `X'` is not considered for its successors.

Furthermore, we implement a rule that prohibits the repetition of the same move or its inverse if the last two actions are identical. Specifically, if we observe a sequence like "`X X`", we avoid repeating move `X` or performing its inverse `X'`. Such repetition would lead to either "`X X X`" (which is equal to `X'`) or simply revert the sequence to `X`, both outcomes failing to advance our search as they merely revisit states encountered just two steps earlier. This approach effectively reduces the creation of replicated states.

Incorporating the Abelian Subgroup of G

An abelian subgroup is a subset of a group where every pair of elements commutes; that is, for any two elements a and b in the subgroup, $ab = ba$. This property is particularly interesting when applied to the Rubik's cube, as it implies a set of moves that can be performed in any order without affecting the outcome.

In the context of G, we identify three significant abelian subgroups, each corresponding to moves on opposite layers of the cube: $UD = \langle\{\texttt{U, D}\}\rangle$ where the subgroup consists of moves on the Up (U) and Down (D) layers. Since these layers are opposite to each other, operations within this subgroup do not interfere with each other. Similarly, we have the abelian subgroups of $FB = \langle\{\texttt{F, B}\}\rangle$ and $RL = \langle\{\texttt{R, L}\}\rangle$ which both groups are generated from the moves on the opposite faces.

To streamline the search process and avoid revisiting previously explored states in solving the Rubik's Cube, our strategy involves identifying and utilizing the longest subsequence of moves that belong to the same abelian subgroup within G. For instance, if we're considering expanding a node with the move sequence "`U D U`", the current algorithm before incorporating restricts us from using `U'` since it only checks the longest identical subsequence from the moves operated. However, another "`U`" would revert the cube to a state previously explored by the sequence "`U' D`" with the commutative property of UD, and similarly, using "`D'`" would take us back to the state achieved by "`U U`". To circumvent this issue, we prioritize sequences that fall within the same abelian subgroup, thereby pruning redundant states.

To accomplish this, we establish a lookup table that correlates each move with its respective subgroup (for example, moves `U`, `U'`, `D`, and `D'` are associated with the UD subgroup). By identifying the subgroup of the last move and then tracing backward through the move sequence, we can count the number of consecutive moves within the same subgroup, thus identifying the longest relevant subsequence. This method effectively helps us in pruning duplicated states, enhancing the efficiency of the search process.

Implementation of the Prevention Algorithm

In implementing the prevention algorithm to refine our search strategy for solving the Rubik's Cube, a critical step involves quantifying the moves within the identified longest subsequence that belongs to the same commutative subgroup. An important consideration throughout this process is that, given the consistent application of the prevention algorithm, the length of this longest subsequence will not exceed four moves. Additionally, it is impossible for a move and its inverse (e.g., `X` and `X'`) to coexist within the same subsequence, as their prior occurrence would negate each other with the commutativity, reverting the cube to a previously encountered state. Therefore, our focus narrows to simply counting the occurrences of identical moves within the subsequence, considering only those moves that align with the subgroup's operations. The pseudocode of the prevention algorithm can be seen in Algorithm 3.

Algorithm 3: GetAllowedMoves

```
Data: $MoveSeq$: Move Sequence
Result: Set of Allowed Moves
1  $AllowedMoves \leftarrow$ All 12 Allowed Moves;
2  if $MoveSeq = \emptyset$ then
3      return $AllowedMoves$;
4  $LastMoveGroup \leftarrow SubgroupDict[MoveSeq[-1]]$;
5  $AllowedMoves \leftarrow AllowedMoves \setminus \{Inverse[MoveSeq[-1]]\}$;
6  $Subsequence \leftarrow$ Longest subsequence that is in $LastMoveGroup$;
7  $PairMap \leftarrow \{\ \}$;
8  for $move \in Subsequence$ do
9      $PairMap[move] \leftarrow PairMap[move] + 1$;
10 for $move \in PairMap$ do
11     if $PairMap[move]\%4 = 2$ then
12         $AllowedMoves \leftarrow AllowedMoves \setminus \{move\}$;
13         $AllowedMoves \leftarrow AllowedMoves \setminus \{Inverse[move]\}$;
14     else if $PairMap[move]\%4 = 1$ then
15         $AllowedMoves \leftarrow AllowedMoves \setminus \{Inverse[move]\}$;
16 return $AllowedMoves$
```

The implemented algorithm incorporates the previously discussed prevention algorithm to enhance search efficiency in solving the Rubik's Cube. This approach allows us to omit up to four previously explored states, significantly optimizing the search process. The "not allowed moves" are summarized in Table 3.2, where `X` and `Y` represent different moves within the same abelian subgroup of the Rubik's Cube group.

The table outlines combinations of `X` and `Y` moves and specifies which subsequent moves are prohibited to prevent revisiting or replicating states.

Table 3.2: Cancellation Steps Using Prevention Algorithm

Move	\	X	X X
\	All Moves Allowed	X'	X, X'
Y	Y'	X', Y'	X, X', Y'
Y Y	Y, Y'	X', Y, Y'	X, X', Y, Y'

By employing the prevention algorithm, significant advancements in search efficiency for solving the Rubik's Cube can be achieved, notably by avoiding the expansion of already explored or replicated states. This strategy is especially powerful when applied to depth-by-depth search methods, such as beam search, where exploring more states is critical.

Comparative Analysis

Implementing the prevention algorithm showcases a significant leap in solving efficiency for the Rubik's Cube, particularly evident through the lens of beam search. To exemplify, let's consider a beam search conducted with a width of 2000 and a depth limit of 26, applied to a randomly generated scramble sequence "`B U' R U U R U D R F R B R' F' U' R' F' F' U R' F' D`". A comparative analysis between executions with and without the prevention algorithm reveals a stark contrast in performance and outcomes.

In the scenario without the prevention algorithm, the search process trawled through 550,621 nodes over 24.84 seconds, yet it failed to uncover a solution. This outcome starkly contrasts with the search incorporating the prevention algorithm, which navigated through **422,682** nodes and successfully identified a 24-step solution within **18.87** seconds. The reduction in the number of explored nodes and the significant decrease in search time—over 5.97 seconds faster—underscore the substantial efficiency gains afforded by the prevention algorithm.

3.3 Proving Invalid States Using Group Theory

Now we want to show that some states of the cube are invalid which can further decrease the search space, i.e., the state cannot be reached by doing regular turns. We will use group theory to prove that. In [Ban82], it was called the "first law of cubology". It goes like this:

Theorem 3.3.1. *A configuration of the Rubik's cube, $S = (\alpha, \beta, \gamma, \delta)$ is valid if and only if the following three conditions are satisfied:*

1. $sgn(\alpha) = sgn(\beta)$,

2. $\gamma_1 + \gamma_2 + \cdots + \gamma_8 = 0 \ mod\ 3$,

3. $\delta_1 + \delta_2 + \cdots + \delta_{12} = 0$ *mod* 2.

Example 3.3.2. For example, the state prevented in Fig. 3.1 is invalid due to a mismatch in parity as the corners remain in their original positions, implying $\text{sgn}(\alpha) = +1$, indicating an even permutation. However, A single swap between edges E_1 and E_3 results in $\text{sgn}(\beta) = -1$, an odd permutation.

Based on Theorem 3.3.1, the valid Rubik's Cube states require consistent parity across both α and β, this discrepancy renders the state invalid.

Example 3.3.3. Take the "Superflipped" as another example; this cube state is renowned for its complexity, being the only known configuration that requires 20 moves to solve under the half-turn metric, as depicted in Fig. 3.13. [RKDD14] It is often referred to as one of the hardest cube states to solve.

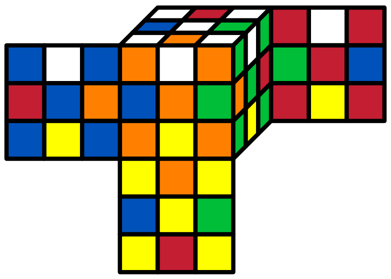


Figure 3.13: Superflipped Cube State

To understand why it constitutes a valid Rubik's Cube state, we examine its permutation and orientation. Initially, all cubies remain in their original slots, rendering $\text{sgn}(\alpha) = \text{sgn}(\beta) = 1$, indicating no permutation. The orientation of corner cubies remains unchanged, as evident by $\gamma_1 + \cdots + \gamma_7 = 0$, which satisfies 0 mod 3. The distinguishing feature of the Superflip is that all edge cubies are flipped, denoted by $\delta_i = 1$ for all $i \in \{1, \cdots, 12\}$, leading to $\sum_{i=1}^{12} \delta_i = 12 = [0]_2$.

Thus, according to Theorem 3.3.1, the Superfliped cube meets all criteria for a valid Rubik's Cube state, despite its complexity and the requirement of 20 steps to resolve under the half-turn metric measure.

Now, let's start to prove Theorem 3.3.1.

Proof. Since it is an if and only if statement, we will prove both sides. Start with the left side, i.e., every possible configuration must satisfy the three conditions.

($\Rightarrow$) Based on the properties explored in Section 3.2.2, every valid state of the Rubik's Cube is attainable through a sequence of allowed moves from the solved state, s_0. So, we will show that the cube state resulting from every fundamental move defined in Definition 2.1.2 satisfies the three properties, and then we are done.

We start with the base case where the cube is fully solved. Under this situation, the state of the cube $s_0 = (\alpha, \beta, \gamma, \delta) = (1, 1, 0, 0)$ where 1 is the identity permutation in S_8 and S_12. And 0's are the zero vectors in a vector space. Therefore, $\text{sgn}(\alpha) = \text{sgn}(\beta) = 1$ whereas $\gamma_1 = \gamma_2 = \cdots = \delta_{12} = 0$ which will lead to the sum of $\gamma = 0$ mod 3 and the sum of $\delta = 0$ mod 2. So, the solved cube state satisfies the three properties.

Let $s = (\sigma, \beta, \gamma, \delta)$ be the cube state that satisfies all the three properties. We want to show that doing any legitimate single move will result in a cube that also satisfies all the conditions, i.e., $s' = m(s)$ satisfies all the three properties for any $m \in \Omega \setminus \{e\}$.

For the first properties, considering doing a move on a Rubik's Cube, we effectively change the positions of exactly four corner cubies and four edge cubies. Notice the σ and τ for all single moves in Table 3.1 are 4-cycles which means that it can be written into 3 transpositions by Theorem 2.2.12. Suppose that we pick any single move $m \in \Omega \setminus \{e\}$ to perform on s with the behavior designed in Section 3.2.1 and denote the resulting cube state as $s' = (\sigma', \beta', \gamma', \delta')$.

By Theorem 2.2.12, $\operatorname{sgn}(\sigma_m) = \operatorname{sgn}(\tau_m) = -1$. We have that

$$\operatorname{sgn}(\alpha') = \operatorname{sgn}(\sigma_m) \cdot (\operatorname{sgn}(\alpha)) = -1 \cdot \operatorname{sgn}(\alpha)$$

Similarly, $\operatorname{sgn}(\beta') = -1 \cdot \operatorname{sgn}(\beta)$. Given that s is a valid configuration where $\operatorname{sgn}(\alpha) = \operatorname{sgn}(\beta)$, the relationships above imply that $\operatorname{sgn}(\alpha') = \operatorname{sgn}(\beta')$. Thus, s', the state after applying move m, satisfies the first property required for valid configurations.

This derivation demonstrates that performing a single move on the cube flips the signs of both $\operatorname{sgn}(\alpha)$ and $\operatorname{sgn}(\beta)$, yet maintains their equality. This ensures that s' remains a valid configuration under the same conditions as the initial state s.

Now, we proceed to validate the second and third properties concerning the orientation changes of cubies as a result of moves applied to the Rubik's Cube. The discussion leverages the behaviors outlined in Table 3.1.

The second property states the $\sum \gamma_i = 0 \bmod 3$. We want to show that doing any move $m \in \Omega \setminus \{e\}$ will maintain this property. According to Table 3.1, moves `U` and `D` do not impact the orientation values of corner cubies; they only permute these values. For the remaining moves (`R`, `L`, `F`, and `B`), we can see that each move precise increments and decrements in the orientation values to make to even. This meticulous adjustment ensures that the total sum of the orientation values of corner cubies remains invariant under all allowed moves. Consequently, the second property is affirmed to hold true across all legitimate move sequences.

Analogously, the third property focuses on the orientation of edge cubies. Observations reveal that moves such as `U`, `D`, `R`, and `L` solely permute the edge orientation values without altering them. This permutation preserves the aggregate sum of edge orientations. For moves `F` and `B`, as delineated in Table 3.1, each move effectively increases or decreases the total edge orientation by an even number. It means doing the move doesn't affect the overall sum of edge orientations. This observation substantiates the validity of the third property.

We have proved that the three properties hold true for each fundamental move and each cube move is composed of fundamental moves, thus, this direction holds true.

($\Leftarrow$) In this direction, we are trying to show that we can obtain any state s that satisfies all three properties from the solved state by doing moves from G. Let $s = (\alpha, \beta, \gamma, \delta)$ and s satisfy the three conditions. The goal is to restore s from s_0 by a move sequence $m \in G$.

First, $\text{sgn}(\alpha) = \text{sgn}(\beta)$, then $\text{sgn}(\alpha) = \text{sgn}(\beta) = 1$ or -1. Let's assume all cases with $\text{sgn}(\alpha) = \text{sgn}(\beta) = 1$ can be achieved. Suppose $\text{sgn}(\alpha) = \text{sgn}(\beta) = -1$ and s_0 be the solved cube, then simply applying one more move $m \in \{\texttt{R, L, U, D, F, B}\}$ to s will make the resulting cube $s' = (\alpha', \beta', \gamma', \delta')$ has the property of $\text{sgn}(\alpha') = \text{sgn}(\beta') = 1$. By assumption, there exists a move sequence $m_1 \in G$ that can obtain s' from s_0. Then we have that

$$m(s) = m_1(s_0)$$

which gives us $s = m^{-1} \circ m_1(s_0)$. This proves that s can be achieved from s_0. For the rest of the proof, we assume $\text{sgn}(\alpha) = \text{sgn}(\beta) = 1$.

Next, we will show that we can achieve α from the solved state $s_0 = (1, 1, 0, 0)$ for all $\alpha \in A_8$ as the first coordinate of s without interference the rest of the coordinates of s. Thus, we show that every $\alpha \in A_8$ can be obtained by a sequence of moves $m \in G$ performed on s_0 with the property that m doesn't change β, γ, and δ in s_0.

Consider the move sequence $M_c =$ `F F L L F' R' F L L F' R F'` designed to create a 3-cycle permutation among the corner cubies in slots 1, 2, and 3. Thus, $\sigma_{M_c} = (1\ 2\ 3)$. Note that $\tau_{M_c} = 1$, $x_{M_c} = 0$, and $y_{M_c} = 0$ by direct computation. So, it means M_c doesn't change the edge permutation and the sum of γ and δ. In other words, M_c only permutes the cubies in slots 1, 2, and 3 and doesn't change the edge permutation and orientation.

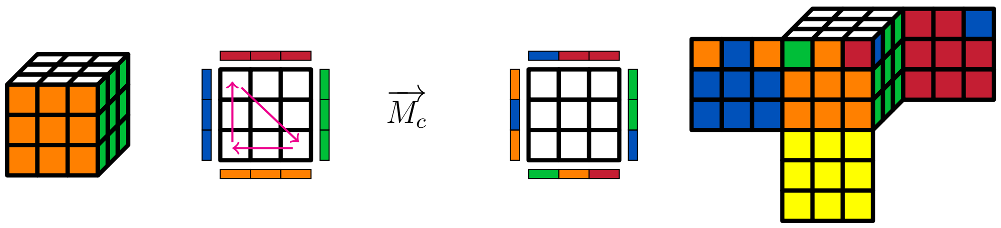


Figure 3.14: 3-cycle permutation of C_1, C_2, C_3

For the cubies at slots 4 through 8, we can construct specific move sequences that permute the cubies at i-th slot to slot 3, without disrupting the cubies in slots 1 and 2. The move sequences are listed in Table B.1. Each m_i in the table represents the move sequence that permutes the cubies at i-th slot to slot 3. Specifically, $\sigma_{m_i} = (i\ 3)$.

Applying the move sequence $M_i = m_i \circ M_c \circ m_i^{-1}$ creates a 3-cycle involving slots 1, 2, and i. The permutation resulting from applying M_i is denoted by $\sigma_{M_i} = (1\ 2\ i)$. Following the procedure, the permutations $(1\ 2\ 3), (1\ 2\ 4), (1\ 2\ 5), (1\ 2\ 6), (1\ 2\ 7), (1\ 2\ 8)$ can be achieved by selecting appropriate i from the set $\{4, \ldots, 8\}$ and M_c. Note that such M_i doesn't change the edge permutations and orientations.

By Theorem 2.2.15, the 3-cycles generate A_8. Therefore, by composing cube moves, we can get all A_8. Thus, we can obtain the first coordinate of s as $\alpha \in A_8$. There must exist a move sequence M_{cp} that sends $s_0 = (1, 1, 0, 0)$ to $s_1 = (\alpha, 1, 0, 0)$

The next step is analogous to the previous step. The goal is to produce $(\alpha, \beta, 0, 0)$ from s_1. With the same procedure, we give a move sequence M_e such that creates a 3-cycle on cubies at slots 1, 2, and 3 without messing the corner permutation and orientations. Specifically,

$$M_e = \texttt{R R U R U R U' R' U' R' U R'}$$

with the effect shown in Fig. 3.15. Similar to restoring the corner cubies, paired

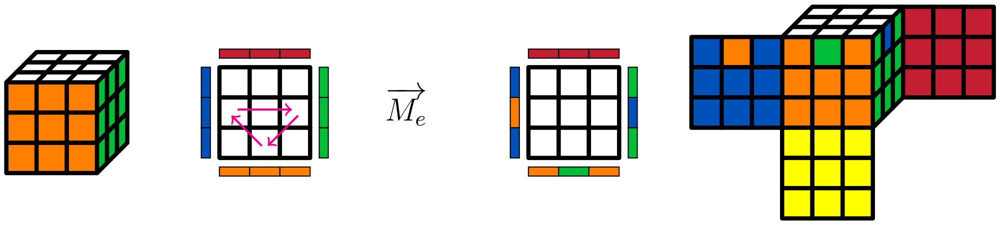


Figure 3.15: 3-cycle permutation of C_1, C_2, C_3

with Table B.2, we can construct specific move sequences that enable permutation among cubies at slots 4 through 12 and the cubies at slot 3, without disrupting the configuration of the cubies in slots 1 and 2.

Applying the move sequence $M_i = m_i \circ M_e \circ m_i^{-1}$ creates a 3-cycle involving slots 1, 2, and i. Notice here that we are not messing up the oriented corner cubies as M_e doesn't change the corner permutations at last and m_i^{-1} cancels out the effect of m_i. Therefore, with the same reasoning, by Theorem 2.2.15, the 3-cycles generate A_{12}. Therefore, there exists an appropriate move M_{ep} that will return all the edge cubies to their home positions without disruption of corner permutations. By composing M_{cp} and M_{ep} in the sequence $M_p = M_{ep} \circ M_{cp}$, we can transition the cube from state s_0 to state $s_2 = (\alpha, \beta, 0, 0)$.

After getting α and β, the next step is to reorient the cubies in s_2 to get s. We start with the corner orientation γ. I will show that for all γ such that $\sum \gamma_i = 0 \bmod 3$ can be obtained by a sequence of move in G.

Note that we can view the sum of all entries in module 3 as a function f such that

$$f : \mathbb{Z}_3^8 \to \mathbb{Z}_3$$

with the operation that $f(v) = [\sum_{i=1}^{8} v_i]_3$ where $v \in \mathbb{Z}_3^8$. Property 2 states that $\sum_{i=1}^{8} \gamma_i = [0]_3$, then the set of possible values of γ is the set of $v \in \mathbb{Z}_3^8$ such that $f(v) = [0]_3$. Recall the definition for the kernel where $\ker(f)$ is the set of vectors that $\{v \in \mathbb{Z}_3^8 \mid f(v) = [0]_3\}$. We can see that the set of all possible values of γ is precisely $\ker(f)$.

Now, if we can find the basis B that spans $\ker(f)$, then for any vector $v_k \in \ker(f)$, v_k can be expressed as the linear combination of vectors in B. Now, the task is to find the basis of $\ker(f)$ and check whether we can find move sequences that generate every linear independent vector in B.

By the rank-nullity theorem, we have that

$$\dim(\ker(f)) + \dim(\mathrm{im}(f)) = \dim(\mathrm{domain}(f)).$$

Since $\dim(\mathrm{domain}(f)) = \mathbb{Z}_3^8 = 8$ and $\dim(\mathrm{im}(f)) = \dim(\mathbb{Z}_3) = 1$, we have that $\dim(\ker(f)) = 8 - 1 = 7$. A basis for the kernel could then be composed of vectors where each has exactly one of the first seven entries as 1, the corresponding entries that sum to zero mod 3, and the rest as 0. Then the following vectors form B for $\ker(f)$.

$$\begin{aligned}
v_1 &= ([1]_3, [-1]_3, 0, 0, 0, 0, 0, 0)\\
v_2 &= ([1]_3, 0, [-1]_3, 0, 0, 0, 0, 0)\\
v_3 &= ([1]_3, 0, 0, [-1]_3, 0, 0, 0, 0)\\
v_4 &= ([1]_3, 0, 0, 0, [-1]_3, 0, 0, 0)\\
v_5 &= ([1]_3, 0, 0, 0, 0, [-1]_3, 0, 0)\\
v_6 &= ([1]_3, 0, 0, 0, 0, 0, [-1]_3, 0)\\
v_7 &= (0, [1]_3, 0, 0, 0, 0, 0, [-1]_3)
\end{aligned}$$

As outlined in Table B.3, each move sequence m_{c_i} is specifically designed to align with one of the basis vectors, denoted as $x_{m_{c_i}} = v_i$ for each $i \in \{1, \dots, 8\}$. The move sequences listed in this table have been meticulously chosen to perform vector additions to γ without permuting the positions of the cubies or altering the orientation of any edge cubies.

Since γ can be represented as a linear combination $\gamma = c_1v_1 + \ldots + c_8v_8$ as v_i's form a basis. The method for determining these scalar coefficients is detailed in Algorithm 15. By applying each vector v_i scaled by its corresponding coefficient c_i, for i ranging from 1 to 8, we can precisely adjust the corner orientations to achieve the desired configuration γ. Let the move sequence M_{co} be designed to generate the specific orientation configuration γ, executing M_{co} will transition the cube from state s_2 to state $s_3 = (\alpha, \beta, \gamma, 0)$.

With the analogous proof to δ by considering the kernel of $h : \mathbb{Z}_2^{12} \to \mathbb{Z}_2$ where $h(v) = [\sum_{i=1}^{1} 2v_i]_{12}$ for some $v \in \mathbb{Z}_2^{12}$. With the same analysis, we can get that the following vectors form the basis to span the kernel.

$$\begin{aligned}
v_1 &= ([1]_2, [1]_2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)\\
v_2 &= ([1]_2, 0, [1]_2, 0, 0, 0, 0, 0, 0, 0, 0, 0)\\
v_3 &= ([1]_2, 0, 0, [1]_2, 0, 0, 0, 0, 0, 0, 0, 0)\\
v_4 &= ([1]_2, 0, 0, 0, [1]_2, 0, 0, 0, 0, 0, 0, 0)\\
v_5 &= ([1]_2, 0, 0, 0, 0, [1]_2, 0, 0, 0, 0, 0, 0)\\
v_6 &= ([1]_2, 0, 0, 0, 0, 0, [1]_2, 0, 0, 0, 0, 0)\\
v_7 &= ([1]_2, 0, 0, 0, 0, 0, 0, [1]_2, 0, 0, 0, 0)\\
v_8 &= ([1]_2, 0, 0, 0, 0, 0, 0, 0, [1]_2, 0, 0, 0)\\
v_9 &= ([1]_2, 0, 0, 0, 0, 0, 0, 0, 0, [1]_2, 0, 0)\\
v_{10} &= ([1]_2, 0, 0, 0, 0, 0, 0, 0, 0, 0, [1]_2, 0)\\
v_{11} &= ([1]_2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, [1]_2)
\end{aligned}$$

To address the reorientation of edge cubies, we employ a similar method as used for the corner cubies, following the guidelines in Table B.4. By executing a sequence of moves, denoted as M_{eo}, we can effectively restore the orientations of the edge cubies to δ. This completes the restoration of the cube's state from s_3 to the desired final state s.

The complete move sequence for achieving this restoration of s from s_0 is:

$$M = M_{eo} \circ M_{co} \circ M_{ep} \circ M_{cp}$$

Thus, all cube states that satisfy the three properties can be obtained from the solved state from a sequence of moves in G. Therefore, the overall statement stays true.

□

Based on Theorem 3.3.1, let's calculate the order of VS which is the set of all valid cube states.

Theorem 3.3.4. $|VS| = |G| = \frac{1}{12}|S| = 43,252,003,274,489856,000.$

Proof of Theorem 3.3.4. As introduced in Section 3.2, the total number of potential Rubik's Cube states, denoted as $|S|$, is calculated based on the 4-tuple representation:

$$|S| = 8! \cdot 12! \cdot 3^8 \cdot 2^{12}.$$

However, this count includes both valid and invalid states. To isolate the number of valid states ($|VS|$), we apply specific conditions from Theorem 3.3.1.

Based on Theorem 3.3.1 and Definition 2.2.14, the number of states must be halved to exclude those with mismatched permutation signs ($\mathrm{sgn}(\alpha) \neq \mathrm{sgn}(\beta)$), as valid cube states require the parity of corner and edge permutations to match. Thus, half of the permutations are discarded, reducing the total number:

$$|s| \to \frac{|s|}{2}.$$

The sum of the corner orientations $\sum_{i=1}^{8} \gamma_i$ must be zero modulo 3 ($[0]_3$). As only one specific orientation configuration is valid when seven cubies' orientations are arbitrarily chosen, this further trims the number of valid states by a factor of 3:

$$|s| \to \frac{|s|}{3}.$$

Similarly, the orientation of the last edge cubie is determined once the orientations of the other 11 edge cubies are set, which reduces the count by another factor of 2:

$$|s| \to \frac{|s|}{2}.$$

Combining these factors, the total number of valid Rubik's Cube states is calculated as:

$$|VS| = \frac{|s|}{2 \cdot 3 \cdot 2}.$$

Since we have show that $|VS| = |G|$ before using the orbit-stabilizer theorem, so we have that

$$|VS| = |G| = \frac{1}{12}|S|.$$

□

Chapter 4

Deep Learning as Heuristic Function

Now, we have shown that the overall validated state of the Rubik's cube is 4.3×10^{19} by Theorem 3.3.4 assuming all the states we met are valid. If we were to store each state, even assuming we could represent each state in a highly compressed format of 10 bytes (which is already optimistically minimal), the total storage required would be:

$$4.3 \times 10^{19} \text{ states } \times 10 \text{ bytes/state} = 4.3 \times 10^{20} \text{ bytes}$$

Converting this to more understandable units:

$$4.3 * 10^{20} \text{ bytes} = 400,000,000 \text{ TB (terabytes)}$$

This amount of data is astronomical and not feasible with today's storage technology, where the largest data centers in the world store only a few exabytes (1 exabyte = 1 million terabytes).

Moreover, even if we assume that accessing any state in our database takes just 1 microsecond, searching through all these states would be extremely time-consuming. For instance, if we needed to search through even a billion (10^9) of these states to find a solution, it would take:

$$10^9 \text{ states} \times 1 \text{ microsecond/state} = 1,000 \text{ seconds}$$

This is over 16 minutes just for a billion searches, and typically, finding a solution would require searching through many more states than this.

Based on the basic analysis above, it is impractical to store all states. In order to approximate the cost-to-solve function (heuristic function), we incorporate neural networks and deep learning to act as an indicator of a certain state.

4.1 Cube Representation for Deep Learning

We have introduced the cubie-level cube representation in Definition 3.1.1, however, since the essence of deep learning is based on matrix manipulation and operation, we need a more appropriate way to represent the Rubik's cube that is more efficient with matrix manipulation. Therefore, we introduce the ***facet-level cube vector***.

Definition 4.1.1. For a given cube state s, the **facet-level cube vector**, denoted by V_s, is a vector with 54 entries. Each element of V_s represents the color of a specific facet of a Rubik's Cube as illustrated in Fig. 4.1. The color at each position in V_s corresponds to a specific facet according to a predefined numbering system, where the numbers 0 through 5 denote colors as follows:

- **0**: Color of the upper (`U`) center cubie.
- **1**: Color of the lower (`D`) center cubie.
- **2**: Color of the left (`L`) center cubie.
- **3**: Color of the right (`R`) center cubie.
- **4**: Color of the back (`B`) center cubie.
- **5**: Color of the front (`F`)center cubie.

Therefore, $V_s \in S_{54}$ represents the permutation of all 54 facets of the cube.

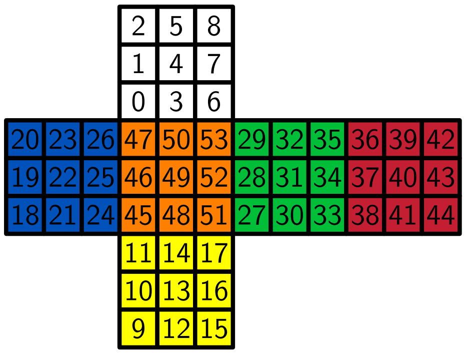


Figure 4.1: Numbering System on the Rubik's cube

For example, let s be the cube presented in Fig. 4.2. We can write V_s based on the color-numbering system presented in Definition 4.1.1 and Fig. 4.1.

$$V_s = \begin{bmatrix} 3 & 3 & 1 & 1 & 0 & 2 & \cdots & 2 & 2 & 1 \end{bmatrix}$$

Definition 4.1.2. ***Facet-level cube string***: A facet-level cube string is derived from a facet-level cube vector V_s. It is a string of length 54 where each character represents the color of a facet at that facet slot. This string is constructed by sequentially concatenating the color values from V_s by the reversed color encoding introduced in Definition 4.1.1.

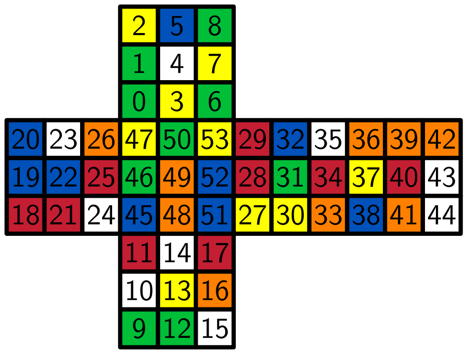


Figure 4.2: The rotations **R** on a solved cube.

With the definition of facet-level cube string, we can write out the cube string as

$$s = RRDDUL \ldots LLD$$

In this representation, each letter in the string s corresponds to a specific facet color on the cube. The initial character, 'R', indicates the color of the first visible facet in the order in Fig. 4.1. Given the color scheme where the front face is orange and the top face is white, in our specific setup, 'R' corresponds to the color green. This matches the visualization depicted in Fig. 4.2.

Now, we redefine the moves in the Ψ using the swaps of facets instead of the 4-tuple definition. For example, the `U` move, `U` permutes facets by shifting the top row of the cube and rotating the adjacent sides appropriately. The mapping for this move could look something like

$$\begin{aligned} \texttt{U} : \{&0:6, 1:3, 2:0, 3:7, 5:1, 6:8, 7:5, 8:2, 20:47, 23:50, 26:53, \\ &29:38, 32:41, 35:44, 38:20, 41:23, 44:26, 47:29, 50:32, 53:35\} \end{aligned}$$

where each pair (i, j) indicates this move sends the color on i-th facet to j-th facet.

With these easier and computationally efficient definitions for the Rubik's cube state and allowed moves, we can develop algorithms and solutions that are easier to implement and optimize, especially for deep learning which is fundamentally matrix manipulation. Therefore, we will implement the algorithm and training model based on the more computationally efficient definitions.

4.2 Deep Neural Network Architecture

The model used for deep reinforcement learning and deep supervised learning described below is based on the model used by [AMSB19].

The input layer of the DNN employs one-hot encoding of the facet-level cube vector to transform it into a one-dimensional format suitable for DNN processing. The network includes six hidden layers. The first hidden layer contains 5,000 neurons, and the second comprises 1,000 neurons. Following these initial hidden layers, there are four residual blocks; each block includes two linear layers with 1,000 neurons each. After each linear layer, Rectified Linear Unit (ReLU) activation is applied to introduce non-linearity, enhancing the network's ability to learn complex patterns. Additionally, batch normalization is implemented following each linear layer to normalize the distribution of the inputs. This process helps stabilize the model during training and mitigates the risk of overfitting.

Residual blocks are structures within neural networks designed to make the training of deep networks feasible and efficient. The key feature of a residual block is the addition of a shortcut connection that skips one or more layers. [HZRS15] These are connections that skip one or more layers and add the input of the residual block to its output as shown in Fig. 4.3. This helps mitigate the vanishing gradient problem by allowing gradients to flow directly through the network during backpropagation.

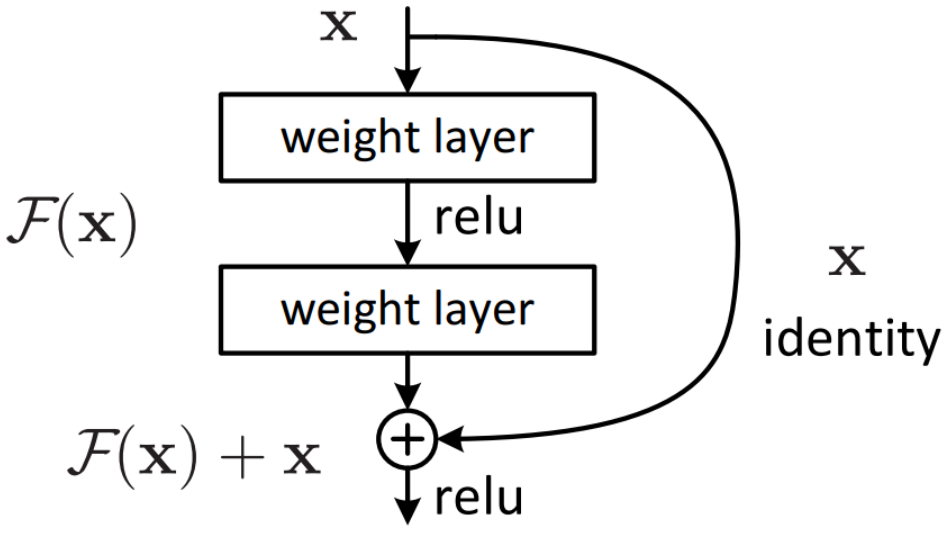

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 4.3: Construction of the Residual Block [HZRS15]

This DNN is tailored for two specific tasks in solving the Rubik's Cube. The first task predicts the minimum number of moves required to solve the cube, employing an output layer with a single neuron. The second task involves predicting the probability distribution of possible moves to optimally advance from the current cube state. For this, the network outputs a 12-dimensional layer that represents logits, which are subsequently converted into a probability distribution using the Softmax function, aligning with the defined allowed moves in the cube's action space.

4.3 Deep Reinforcement Learning Strategy

This section outlines the value iteration approach to solve the MDP defined in Section 2.4.3 inspired by [AMSB19] based on solving π^* from V^*.

4.3.1 Value Iteration

Value iteration leverages Bellman's optimality principle to iteratively refine estimates of the optimal value function V^* through approximate dynamic programming.

As discussed in Section 4.3, the Bellman optimality equation for value iteration is formulated as:

$$V^*(s) = \max_\pi \mathbb{E}_\pi \left[R_0 + V^*(s')\right]$$

First, we rewrite V^* as

$$V^*(s) = \max_a \sum_{s'} P(s'|s,a)(R(s',s,a) + V^*(s'))$$

Here, $\max_a$ denotes the maximization over all possible actions a, reflecting the core idea of dynamic programming without explicitly averaging over policies.

In a deterministic environment of the Rubik's Cube, where $P(s'|s,a) = 1$ for the next state s' resulting from taking action a in the state s, the equation simplifies to:

$$V^*(s) = \max_a (R(s',s,a) + V^*(s'))$$

This represents the direct computation of the value function under deterministic transitions, emphasizing the maximum reward achievable through the best action.

We can approximate the optimal state-value function V^* using a Deep Neural Network (DNN) with parameters θ, denoted as $\hat{V}(s,\theta)$. This function approximation approach is crucial when the exact computation of V^* is infeasible:

$$\hat{V}(s,\theta) = V^*(s)$$

The approximation $\hat{V}(s,\theta)$ is iteratively refined through training, where the Mean Squared Error (MSE) between $\hat{V}(s,\theta)$ and $\hat{V}(s',\theta)$ for transitions to subsequent states s' drives the optimization. This training utilizes standard forward and backward propagation techniques as described in Chapter 2.

With $\hat{V}(s,\theta)$, the optimal policy π^* can be derived, which dictates the best action a to take in each state s:

$$\pi^*(s) = \arg\max_a \left[R(s',s,a) + \hat{V}(s',\theta)\right] \tag{4.3.1}$$

This equation outlines how the policy is extracted from the approximated value function, focusing on maximizing the expected return from each state.

4.3.2 Training Policy For DRL

For this section, we are going to implement the training process from the theoretical framework described in the previous section as shown in Algorithm 4.

It begins by initializing the network parameters, denoted as theta, which are set using the $InitialParameters()$ function. This function provides initial zero values for the parameters. A copy of these initial parameters is stored in θ_e used for evaluation or as a temporary snapshot during the training process.

Algorithm 4: TrainingDRLModel

```
   Data: $MaxSteps$, $B$: Batch Size, $MaxDepth$, $\epsilon$: error threshold
   Result: $\theta$ : Trained neural network parameters
1  $\theta \leftarrow InitialParameters()$;
2  $\theta_e \leftarrow \theta$;
3  for $step \in MaxSteps$ do
4      $X \leftarrow GetSrambledCubes(B, MaxDepth)$;
5      for $x_i \in X$ do
6          $y_i = \max_a(R(x'_i, x_i, a) + \hat{V}(x'_i, \theta_e))$;
7      $loss \leftarrow MSE(Y, \hat{V}(X, \theta))$;
8      Zero Out the gradients in $Adam$;
9      Perform Backpropagation;
10     Update $Adam$;
11     if $loss \leq \epsilon$ then
12         $\theta_e \leftarrow \theta$;
13 return $\theta$;
14 Function GetSrambledCubes($B, MaxDepth$)
15     for $depth \in MaxDepth$ do
16         for $i \in B$ do
               /* Randomize Cube $Depth$ steps; Algorithm 3 applied here */
17             $X \leftarrow RandomizeCube(depth)$;
18     return $X$;
```

Training proceeds in a loop at line 3, executing for a maximum number of iterations defined by $MaxSteps$. In each iteration, the algorithm processes a batch of scrambled cubes. These cubes are obtained through the $GetScrambledCubes(B, MaxDepth)$ function defined in Reference, where B is the batch size and $MaxDepth$ is the maximal scrambling depth. Note that the prevention algorithm (Algorithm 3) is used here to make the scrambled moves don't cancel themselves out. This can make the training set more evenly spread.

From lines 5 to 6, for each cube in the batch, the algorithm calculates a target value y_i. This target value is the maximum of the sum of the immediate reward and the estimated value function for the next state and action, calculated as $R(s', s, a) + \hat{V}(s', \theta_e)$.

The key to the training process is the loss calculation and subsequent update of the network parameters from line 7 to line 10 in Algorithm 4. The Mean Squared Error (MSE) between the predicted values from the network $\hat{V}(s, \theta)$ and the target batched values Y is computed to assess the accuracy of the network predictions. The gradients are then reset in the Adam optimizer and the network parameters are updated based on these gradients through backpropagation

An important step in the training loop is the convergence check at lines 11 and

12. If the calculated loss is less than or equal to a predefined error threshold ϵ, the algorithm updates θ_e with the current values of θ, suggesting an improvement in the network's performance. The loop continues until the maximum number of iterations is reached or the desired accuracy level is achieved.

Finally, the algorithm concludes by returning the trained network parameters θ, which are optimized to solve the scrambled cubes.

We refer to the developed neural network as the DRL model, which is capable of calculating the minimal number of steps required to solve a Rubik's Cube from any given state. For any cube state s, we define the function ComputeMinSteps(s) to encapsulate the process whereby the DRL model predicts the minimal number of steps needed to solve the cube starting from state s.

4.3.3 Validation of the DRL Model

The DeepCubeA dataset comprises 1,000 optimal solutions for solving various cube states. [AMSB19] By reversing the optimal solution, we effectively generate both the scrambled cube as our data and the scramble depth as the label, facilitating model validation.

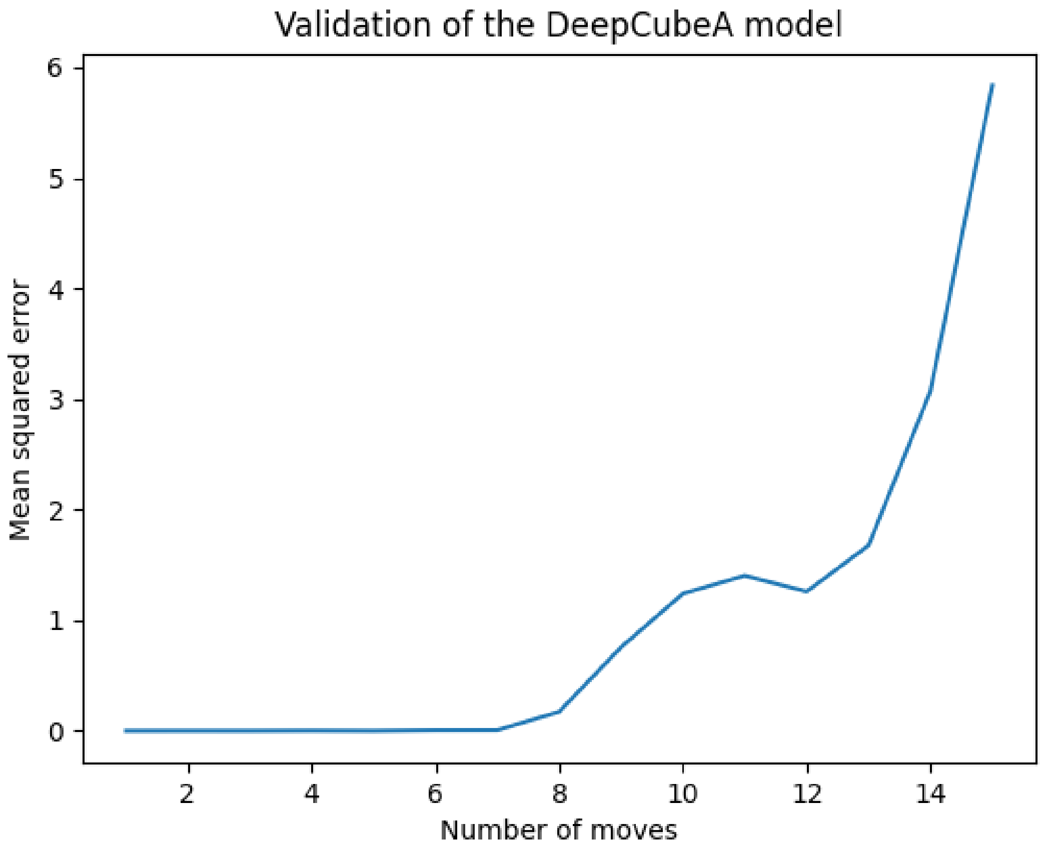


Figure 4.4: The MSE Loss of the DRL Model Validation

We have selectively sampled scrambles from these 1,000 solutions, varying their lengths between 1 and 14 moves. Given that these sequences are derived from optimal solutions, each segment, regardless of length, also represents an optimal solution. Therefore, the assigned labels are accurate reflections of the true parameters. We assess the performance of the DRL model by comparing its predictions against actual scramble depths, analyzing the Mean Squared Error (MSE) across depths ranging from 1 to 14. The results are depicted in Fig. 4.4.

The analysis shows an increase in MSE loss for scramble depths of 10 and beyond. This suggests that the model's accuracy diminishes as the scramble depth increases, indicating challenges in accurately predicting further scrambled states. To address this, we propose a supplementary model to uncover more nuanced features of the cube's state. This auxiliary model will enhance our primary heuristic function by providing additional information on the proximity of a cube state to being solved.

4.4 Supportive Supervised Learning Model For Enhanced Heuristic Function

In light of the limitations and inaccuracies observed in the DRL model, we incorporate another supervised learning model to learn another aspect of the Rubik's Cube. From Theorem 3.3.1, we have that for every valid cube state s, there must exist a move sequence M in G such that $M(s_0) = s$ where s_0 is the solved state. Then, given that such a move sequence must exist, we can find M^{-1} to reverse s back to s_0. This model predicts the inverse of each move in M iteratively, effectively applying M^{-1} to unscramble s back to the solved state s_0, a process referred to as unscrambling in the literature [Tak23]. Intuitively, we predict the move that has the higher chance of making the cube closer to s_0.

For example, if the cube state is obtained by applying the move `R U F'` to s_0, then the model should predict a probability distribution $\hat{P}$ such that $\hat{P}_F = 1$ and all other moves have a probability of 0.

This new model augments the information provided by the DRL model's state-value $\hat{V}(s, \theta)$, enriching the data available for guiding heuristic searches. The integration of this model with the DRL model will be further discussed and analyzed in Section 4.5.

To restate, given a cube state s, our objective is to approximate a function $\hat{f}(s) = P(A)$ that accurately represents $P(A)$, where $P(A)$ is the probability distribution across all possible actions. This distribution quantifies the likelihood of each action being the optimal next step, offering a systematic approach to decision-making in the unscrambling process.

4.4.1 Training Policy For DSL

This approach is a typical self-supervised learning scheme where the label is generated from the data during the generation steps, effectively transforming an unsupervised learning task into a supervised learning problem. Traditionally, direct access to labels from the data isn't possible in unsupervised settings like the Rubik's Cube.

To generate both data and its corresponding label, we scramble the cube randomly and then record the inverse of the last move performed. Mathematically, let $M \in G$ such that $M = M_1 \circ M_0$ where $M_1 \in \Psi$. Then, he data is represented by $s' = M(s_0)$ and the label is M_1^{-1}. This methodology underpins the Data Generating Algorithm, detailed in Algorithm 5.

In Algorithm 5, we establish the quantity of data and labels to be produced at each scramble depth B, extending up to a defined maximum depth, $MaxDepth$. This structured approach ensures an even distribution of the generated data. During the execution of each move, we effectively utilize the prevention algorithm (referenced as Algorithm 3) to negate any effects from previous moves. This careful cancellation helps in making the data more sparse and evenly distributed, which is crucial for enhancing the model's performance during the training phase.

Algorithm 5: DataGenerator

```
Data: $B$: Batch Size Per Depth, $MaxDepth$: Max Scrambling Depth
Result: Batched Data: Cube States with the Label
1 $States \leftarrow \{\}$;
2 $Label \leftarrow \{\}$;
3 for $i \in \{1, \ldots, B\}$ do
4     for $step \in \{1, \ldots, MaxDepth\}$ do
5         $NewState, LastStepDone \leftarrow cube.randomize(step)$;
6         $NewLabel \leftarrow InverseMove[LastStepDone]$;
7         $States \leftarrow States \cup \{NewLabel\}$;
8 return $\{States, Label\}$;
```

To illustrate how the data and labels are generated in this self-supervised learning framework, consider a Rubik's Cube state s obtained by applying the move sequence `U R F`, as depicted in Fig. 3.10. In this case, the label would be a probability distribution P where $P_{F'} = 1$ (indicating the move to reverse the last move in the sequence), and 0 for all other possible moves. The data, represented as a facet-level cube vector (Definition 4.1.1), is explicitly shown below:

$$V_s = \begin{bmatrix} 2 & 0 & 0 & 2 & \cdots & 1 & 3 & 3 \end{bmatrix}$$

This vector $s = V_s$ is then fed as input data $\boldsymbol{X}$ into the DNN described in Section 4.2. The model is tasked with predicting a probability distribution $\hat{P}$ across the action space. To optimize the model, we employ the Cross-Entropy Error (CEE) as a loss function to measure the discrepancy between the actual probability distribution P and the predicted distribution $\hat{P}$. And we use *Adam* as the optimizer to update the parameter in DNN.

This transformation from an unsupervised to a supervised learning problem allows us to apply conventional neural network training techniques, such as forward propagation and backward propagation, as discussed in Section 2.4.2. The details and implementation of the training algorithm are further elaborated in Algorithm 6. This structured approach ensures systematic learning and adaptation, leveraging the rich data from the cube states to train the model effectively.

Algorithm 6: TrainingModel

```
   Data: $MaxSteps$, $Model$ : Model defined in Ref, $B$, $MaxDepth$
   Result: Batched Data: States with the Label
1  Set $Model$ to training mode;
2  $LossFunction \leftarrow CrossEntropyLoss$;
3  $Optimizer \leftarrow Adam$;
4  for $i \in \{1, \ldots, MaxSteps\}$ do
5      $Data \leftarrow DataGenerator(B, MaxDepth)$;
6      Reshape $Data.X$ and $Data.Y$ for $Model$;
7      Fed $Data.X$ and $Data.Y$ to Device;
8      Predict $Pred_Y$ using $Model$ with $Data.X$;
9      $loss \leftarrow LossFunction(Data.Y, Pred_Y)$;
10     Zero out the gradients in $Optimizer$;
11     Perform backpropagation;
12     Update $Optmizer$;
13 return Trained $Model$;
```

In this algorithm, we utilize the Cross Entropy Loss (CEE) to measure the discrepancy in probability distributions and employ the Adam optimizer for optimizing the network during backpropagation, as specified in lines 2 and 3. Throughout the training process, up to the $MaxSteps$, we continuously generate batches of data and labels using the Data Generating Algorithm (Algorithm 5). The forward propagation is conducted in lines 7 and 8, while lines 9 to 12 focus on backpropagation and updating the optimizer with the new parameters.

We set $MaxSteps$ to 2,000,000 and $MaxDepth$ to 26, based on God's number [KC07]. At each depth, $1,000$ states and labels are generated, leading the model to process approximately

$$2,000,000 \times 26 \times 1,000 = 52,000,000,000 \text{ states}$$

during training, which represents about $1.23 \times 10^{-7}\%$ of the Rubik's Cube state space. Notably, some states, especially at depths 1 and 2, are replicated due to the limited number of unique configurations available at these lower depths.

Throughout the training, the CEE loss is monitored and plotted, as illustrated in Fig. 4.5.

Toward the end of the training, fluctuations in the loss curve are be observed, potentially due to the exponential scaling of the x-axis, causing losses to cluster and creating an illusion of "disturbance". This concludes our discussion on training the DSL model to predict the moves to unscramble the cube by estimating the inverse of the last move that generated state s.

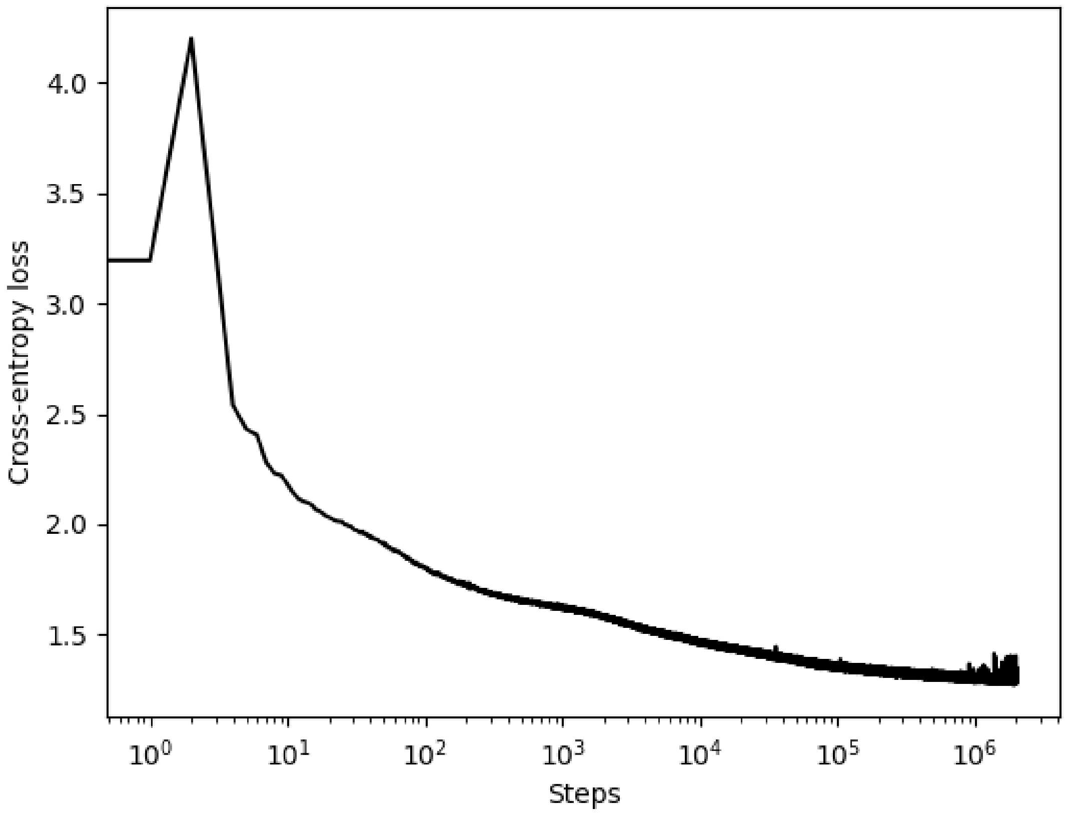


Figure 4.5: Training Loss Curve

We refer to the trained neural network as the DSL model. For any cube state s, we define the function ComputeProbDistribution(s) to encapsulate the process by which the DSL model predicts the probability distribution over the action space based on s indicating the likelihood of each move being the optimal current action.

4.4.2 Validation of the DSL Model

We adopted the same procedure to measure loss at each scramble depth, assessing how the model performs across various levels of cube complexity as shown in Fig. 4.6.

As illustrated in Fig. 4.6, the model's performance deteriorates at scramble depths of 10 and beyond, where the loss increases significantly. This trend is similar to the challenge faced by the DRL model, as discussed in the previous section, where both models struggle to provide accurate predictions for highly scrambled states—typically those requiring more than 14 moves to solve.

However, at scramble depths of 10 and below, both the DRL and DSL models demonstrate a high degree of accuracy in their predictions compared to the true values. This indicates that while the models are effective at handling moderately scrambled cubes, their reliability decreases as the complexity of the cube's state in-

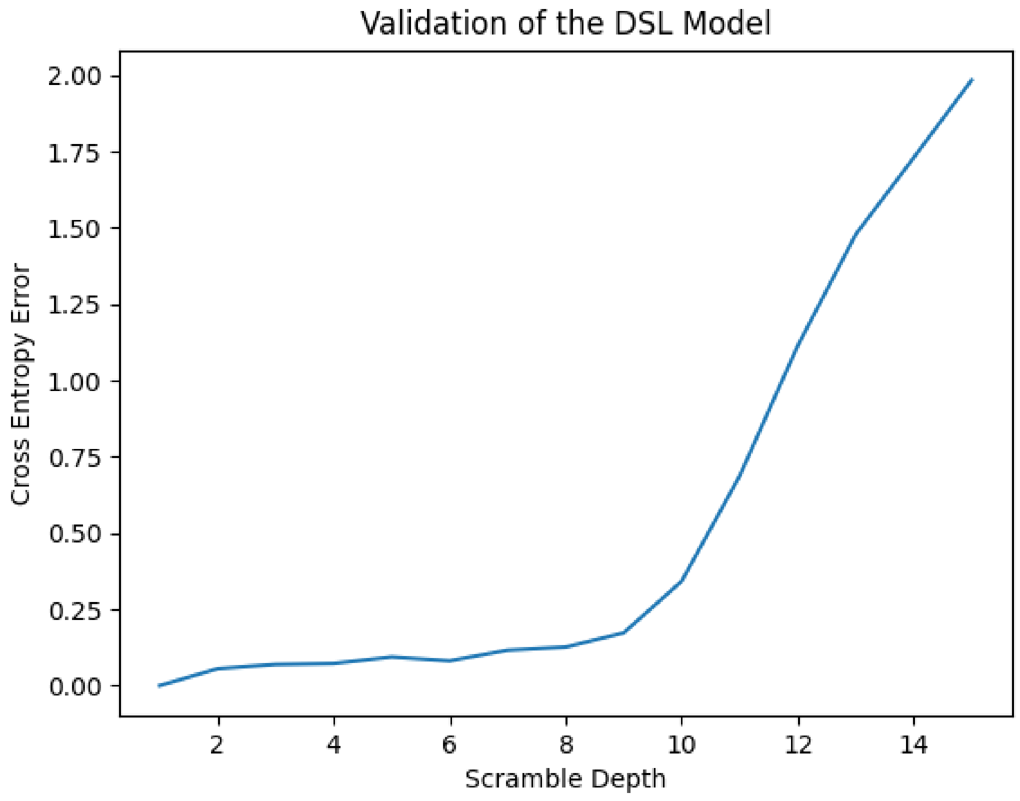


Figure 4.6: Validation of the DSL Model

creases.

4.5 Combining Two Models

In the development of the DSL and DRL models, we can combine them to measure the heuristic value of the current cube state. By leveraging the batch processing capabilities inherent in DNN structures and incorporating a prevention algorithm (Algorithm 3), we have designed Algorithm 7 to compute the fitness of successors for a set of cubes.

The Algorithm 7 algorithm begins by receiving a set of cube states that need to be expanded to find their successors. Suppose there are N such states. We input these N best cube states into the DSL model to predict the probability distribution of the best move that could bring each closer to the solved state. The input dimension for this operation is $N \times 324$ since we are using the one-hot encoding (the dimension of a single cube state is 1×324—reflecting 54 cube facets and 6 colors). Following the retrieval of the predicted probability distribution p, node expansion begins. Each node in the set of N best cubes undergoes allowed moves, filtered through the prevention algorithm to ensure only effective moves are applied.

Algorithm 7: ComputeFitness

```
  Data: $BestNodes$ : Nodes to be expanded
  Result: a list of probability distributions $p$ and a list of values $h$
1 $p \leftarrow ComputeProbDistribution(BestNodes)$; $Batch \leftarrow [\,]$;
2 for $node \in BestNodes$ do
3     $AvailableMoves \leftarrow GetAllowedMoves(node.moves)$;
4     for $move \in AvailableMoves$ do
5         Apply $move$ on $node.state$ to get $NewState$;
6         $Batch \leftarrow Batch \cup NewStatet$;
7 $h \leftarrow ComputeMinSteps(Batch)$;
8 return $p, h$
```

All successors are added to a *Batch*, and the minimal steps needed to solve each are calculated using the DRL model; these scores are stored in h. We then return the probability distribution and the minimal steps, where p represents the probability distribution over the *BestNodes*, and h estimates the minimal steps required to solve the cube for the successors of the *BestNodes*.

For instance, consider a cube state s scrambled with the sequence `U R F` shown in Fig. 4.7 In this case, we expect reverse move `F'` to have the highest probability based on the DSL model's predictions. Consequently, the state resulting from applying `F'` should have the lowest fitness (estimated steps to solution) among all states derived from s. Then, the final fitness will be the $f(x) = g(x) + h'(x)$ where $h' = h(x) - p(x', m)$ and x is the cube state obtained by doing m to x'.

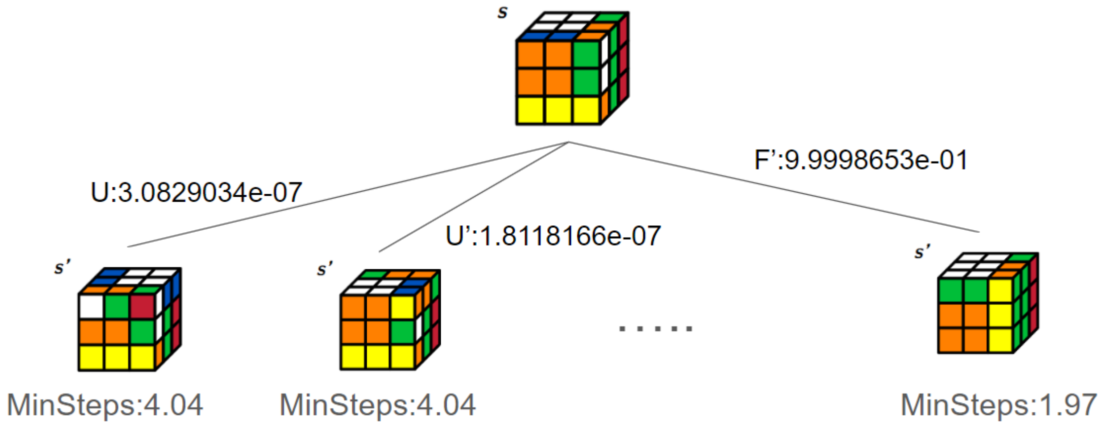


Figure 4.7: Example of Using Multi-Heuristic Function

Previously, the DRL model alone was used to measure the minimal steps required to solve the cube, guiding the search as implemented in [AMSB19]. However, testing revealed it to be largely inadmissible, failing in 37.83% of cases when generating cubes at any scramble depth less than 26 steps, where $\hat{h}(s) > h^*(s)$, though it maintained consistency by failing only 0.51% of the time.

With the supporting DSL model, we evaluate the cube state with prior knowledge of its parent state. For example, if a cube state s has a probability distribution predicted by the DSL model indicating $P(s)_U = 1$ and 0 elsewhere, and we consider the successor state s' resulting from applying U to s, in addition to producing the DRL score for s', we can also consider $P(s)$. A higher probability in $P(s)$ implies a greater likelihood that the move with the highest probability is the one that will bring the cube closer to the solved state. Therefore, subtracting $P(s)_U$ from $H(s')$ modifies the original value $H(s')$ with this prior knowledge.

By incorporating this multi-heuristic approach to assess proximity to the solved state, we leverage additional features and knowledge about the state to guide the search. This method significantly reduces the admissibility failure rate from 37.83% to 9.82% while maintaining a consistent error rate, enhancing the search's accuracy and speed compared to using the DRL model alone. This improvement prevents misdirected searches due to inadmissibility, making the system more efficient and effective.

Chapter 5

Integration and Implementation

In this chapter, our objective is to synthesize the various components discussed in the earlier chapters, crafting algorithms tailored to meet the specific goals outlined in Section 1.3. The full Python implementations can be found here.

To achieve these goals, we will enhance the A* search algorithm and the beam search algorithm initially introduced in Section 2.3 with structural and conceptual improvements. We will incorporate all the enhancements previously described, specifically Algorithm 3 and Algorithm 7.

The culmination of this integration will be presented through a user-friendly interface that includes comprehensive features along with visualization capabilities. This solver will enable users to customize their cubes using facet-level cube string (Definition 4.1.2), apply random scrambles, and choose from different search algorithms and parameters to solve the cubes according to varying objectives.

Central to these search algorithms is the consistent structure of the nodes, which form the backbone of the search process. Each node in the search framework is defined as Definition 5.0.1.

Definition 5.0.1. A node is a 6-tuple structure $n = (m, cube, g, h, f, solved)$ which is defined as:

- **Move Sequence (m):** Records the sequence of moves applied to the scrambled cube to reach the current state, tracing the path from the initial to the current configuration.
- **Cube State ($cube$):** Represents the current configuration of the cube using the facet-level cube vector (Definition 4.1.1).
- **Distance from the Start State (g):** Denoted as $g(x_0, c)$ where x_0 is the initial scrambled cube and c the current cube state, measuring the number of moves from the starting state to the current state.
- **Heuristic Value (h):** A score estimating the node's closeness to the goal state from the *ComputeFitness* algorithm (Algorithm 7).
- **Fitness Value (f):** Calculated as $(g(x_0, c)+h(c))$, combining g and h to assess the node's potential and guide the prioritization of node expansion.

- **Solved Index** (*solved*): Indicates whether the cube is solved, providing a straightforward way to determine the termination or continuation of the search.

Before delving deeper into the implementation of the search algorithms, it is important to address the ability for users to input their own facet-level cube configurations. This customization necessitates the development of a validation algorithm to ensure the integrity and solvability of user-defined cube states.

5.1 Validation Algorithm for the Cube States

One aspect of the solver is that it allows the user to configure their cube states using the facet-level cube string. Users will enter the facet-level cube string in the solver and we need to verify the state configured from the string is a state that can be solved with allowed moves based on Section 3.2.2. This means we need to create an algorithm to transfer from facet-level cube string to cubie-level cube state, and then use the three conditions to verify the states.

5.1.1 Cubie Representation and Dictionary Setup

In this section, we'll explore the foundational checks and structures required to validate Rubik's Cube states represented by the string s. The initial and fundamental check involves verifying that s consists of 54 characters, which aligns with the number of facets on a Rubik's Cube. This check ensures that s could potentially represent a valid cube state within the set S, defined as the set of all possible cube states (Section 3.2.1).

To further facilitate the validation and manipulation of cube states, we define two dictionaries, *CornerCubies* and *EdgeCubies*, which map each cubie to its respective facets. For instance, a corner cubie C_1 consists of facets `U F R` as defined in Section 3.1. The sequence in which the facets are listed is important; it indicates the orientation of each cubie based on a predefined encoding following the procedure described in Section 3.1. For example, we assign the number 0 to color `U`, 1 to `F`, and 2 to `R` in Section 3.1, then facets of C_1 in *CornerCubies* follow this order. We do the same for *EdgeCubies*.

$$\begin{aligned} CornerCubies &= \{C_1 : \{\mathtt{U}, \mathtt{F}, \mathtt{R}\}, \ldots, C_8 : \{\mathtt{D}, \mathtt{B}, \mathtt{R}\}\} \\ EdgeCubies &= \{E_1 : \{\mathtt{U}, \mathtt{R}\}, \ldots, E_{12} : \{\mathtt{B}, \mathtt{R}\}\} \end{aligned}$$

Furthermore, we construct two additional lists, *CornerSlots* and *EdgeSlots*, to relate specific positions within string s to the cubie slots. Each slot corresponds to a particular set of positions within the string that denote the facets of the cube. For instance, based on Fig. 4.1, Slot 1 for the edge cubies consists of indices 7 and 32. Therefore, the 7th and 32nd characters of s reflect the colors of the cubies located at Slot 1.

The concatenated matrix from Fig. 4.1 is represented in s, where each character corresponds to a specific facet's color. By extracting characters from these positions in s, we determine if they can be matched to any defined cubies regardless of their arrangement. If not, the string s does not represent a valid cube state. The positions for corners and edges are defined as follows:

$$CornerSlots = \{[6, 53, 29], \dots , [15, 36, 33]\}$$
$$EdgeSlots = \{[7, 32], \dots , [37, 34]\}$$

For example, if the cube state is configured as Fig. 4.2, then $EdgeSlots[0] = [7, 32]$ represents the Slot Edge 1 defined in Section 3.1. We take the 7th and 32nd characters from s which is DL that matched $E_6 \in EdgeCubies$. Then we continue with the rest of the cubies.

Algorithm 8: Parity

```
Data: $Perm$: a permutation
Result: True if the permutation has even parity, False otherwise
1  $length \leftarrow$ length of $Perm$;
2  $elementsSeen \leftarrow$ an array of size $length$, initialized to $False$;
3  $cycles \leftarrow 0$;
4  for $index \leftarrow 0$ to $length - 1$ do
5      if not $elementsSeen[index]$ then
           /* If the index hasn't been seen */
6          $cycles \leftarrow cycles + 1$ ;          // Start a new cycle
7          $current \leftarrow index$ ;  // Start tracing the cycle from the current index
8          while not $elementsSeen[current]$ do
               /* Loop until cycle is closed */
9              $elementsSeen[current] \leftarrow True$ ;    // Mark current index as seen
10             $current \leftarrow Perm[current]$ ;    // Move to the next index in the cycle
11 return $(length - cycles) \mod 2 = 0$ ;  // Even parity if true, odd if false
```

5.1.2 Validity Checks

After verifying that the string s represents a valid set of cube states in S, we proceed to transform s into a cubie-level representation. This transformation is essential for applying Theorem 3.3.1. The process involves reconstructing the permutation α by systematically checking each slot's configuration and assigning it back to its respective cubie in the defined order.

For example, if we find that C_5 is present in corner slot 1, the first coordinate of α is assigned as C_5. This procedure is iteratively applied to both α and β to complete the permutations. With the assistance of algorithms such as those outlined in Algorithm 8, we can determine the parity of both corner and edge permutations.

If these permutations do not conform to the conditions set forth in Theorem 3.3.1, further processing is unnecessary, and the cube state can be deemed invalid.

For cubie orientation, we use a similar methodology. During the permutation checks, the orientation for each slot is assessed by examining the color at the first coordinate of the slot and noting the index of this color in the original cubie dictionary. For instance, if E_9 occupies slot 1 of the edge slots and the first color of this slot (position 7 in s) is `R`, we look up `R` in E_9 within *EdgeCubies*, which may correspond to an index of 1. Thus, the first coordinate of the orientation vector δ would be set to 1. This process is repeated for all corners and edges. In this way, by inputting a facet-level cube string s, we can verify the validity of the constructed cube using *Algorithm* 9.

Algorithm 9: Validation Algorithm

```
Data: $s$: facet-level cube string
Result: True or False
1  Initialize $CornerCubies$, $EdgeCubies, CornerSlots, EdgeSlots$;
2  Initialize $CornerPerm = \{\}, EdgePerm = \{\}, CornerOri = 0, EdgeOri = 0$;
3  for $corner \in CornerSlots$ do
4      Check colors at positions indexed by $corner$;
5      $C_i \leftarrow$ The corner cubie corresponding to the colors;
6      if $No\ Such\ C_i$ then
7          return False;
8      $CornerPerm \leftarrow CornerPerm \cup \{C_i\}$;
9      $FacingColor \leftarrow$ the color at $corner[0]$;
10     $ColorIndex \leftarrow$ The index of $FacingColor$ in $CornerCubier[C_i]$;
11     $CornerOri \leftarrow CornerOri + ColorIndex$;
12 Do the same for edges;
13 if $CornerOri\ mod\ 3 = 0\ and\ EdgeOri\ mod\ 2 = 0\ and$
    $Parity(CornerPerm) = Parity(EdgePerm)$ then
14     return True;
15 return False;
```

If at any point the orientation and permutation checks fail to meet the requirements of the second and third properties from Theorem 3.3.1, the cube state is invalidated. If, however, s passes all these tests, it is confirmed as valid. The next step involves constructing the facet-level cube vector from s, which is then ready to be utilized in search algorithms. This structured approach ensures that only configurations that faithfully represent real Rubik's Cube states are advanced for algorithmic solving, thereby maintaining the integrity and efficiency of the computational solving process.

5.2 Multi-heuristic Batched Weighted A* Search Algorithm (MBWA*)

This search algorithm is designed to optimize search speed while maintaining solution quality within an acceptable range. MBWA* builds on the foundational principles of the A* search algorithm. Due to the heuristic function h being non-admissible, as discussed in Section 4.5, relying solely on a traditional A* search does not guarantee an optimal solution, even with substantial computational resources. To address these challenges, we have introduced several structural and conceptual improvements to enhance the A* search methodology.

5.2.1 Conceptual Improvements of MBWA*

Weighted A* Search

The conventional A* search algorithm prioritizes nodes based on the function $f(x) = g(c, c') + h(c', x)$, placing equal importance on solution quality and search time. However, this approach demands significant computational resources. The use of an inadmissible heuristic h can further complicate the search, as it may overestimate the cost to reach the goal, leading the search in suboptimal directions. Thus, traditional A* may not always represent the most efficient strategy for finding optimal solutions.

By increasing the weight on h, the algorithm biases the search towards states that appear closer to the goal, potentially reducing search time at the expense of achieving a suboptimal solution. This strategy significantly reduces both the time and memory required for the search by favoring states nearer to the solved state and de-emphasizing the path taken thus far. Heavy reliance on h, akin to Greedy Best-First Search (GBFS) that depends solely on the heuristic, is not viable for two reasons: firstly, the inadmissible nature of h could misguide the search, potentially leading to loops or incorrect paths; secondly, as the search is constrained by solution length, the steps taken so far remain relevant. Thus, identifying the appropriate balance in the weighting of h is crucial to enhancing search efficiency without compromising too heavily on solution quality. [ED09]

Batched Nodes To Expand

As indicated in Section 2.4.2, since the heuristic functions we got from Chapter 4 are DNNs, we can apply batch processing to search algorithms, particularly when evaluating the fitness of successors. All successors can be processed in a single batch through the DNN, with indices used to retrieve specific results. This approach not only accelerates the search process but also places higher demands on GPU resources due to the increased memory requirements of large matrix multiplications compared to simple vector operations. However, since the primary objective is to minimize computation time, batch processing is an effective strategy for both training and searching.

Based on Fig. 5.1, the time required to process data increases linearly with the number of cube states being computed. However, the computation time remains consistently around 1 second, regardless of the batch size. Thus, it is possible to explore more nodes within the same time frame, increasing the likelihood of identifying a suboptimal solution more rapidly and with potentially better quality.

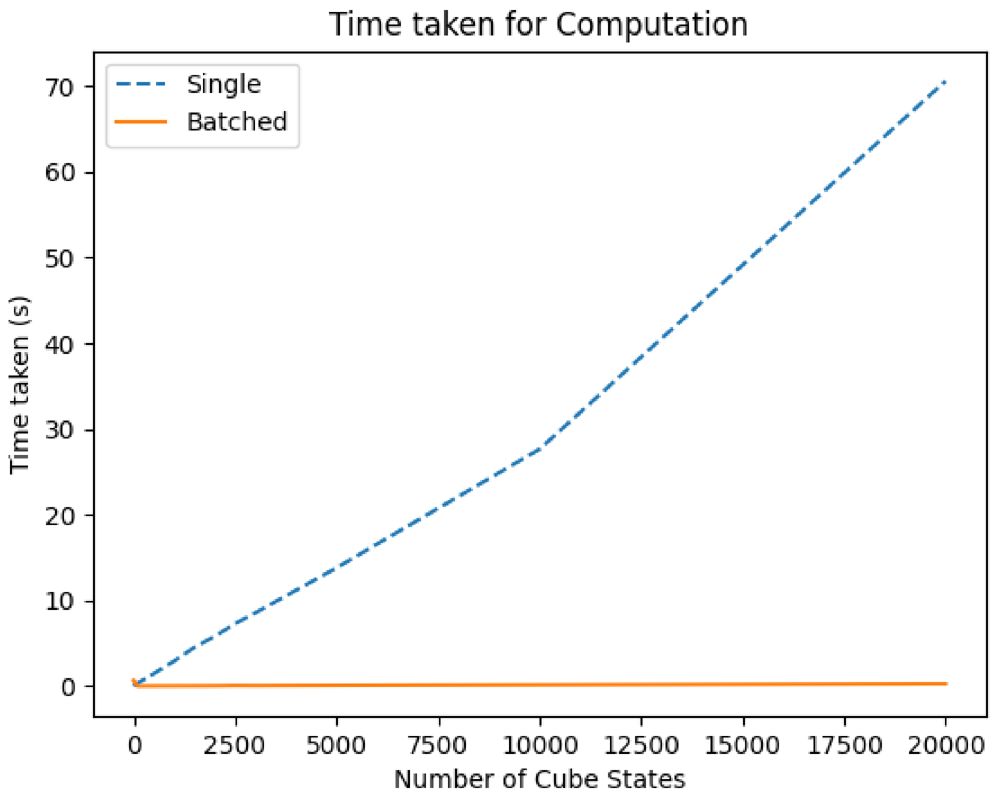


Figure 5.1: Time Usage of Batch Processing Compared with Single Processing

The primary drawback of batch processing is that it theoretically expands more nodes than traditional A* search, resulting in higher memory usage as it stores multiple node states simultaneously. However, prioritizing time over memory consumption may justify this trade-off, allowing for a broader exploration rate within the same time span to achieve quicker and potentially better solutions.

5.2.2 Structural Improvements of MBWA*

Priority Queue To Store Open Nodes

We employ a priority queue to manage the nodes in the open list. This data structure keeps the node with the lowest f value at the forefront, ensuring efficient access. When extracting the best N nodes from the open list (as specified in line 6 of Algorithm 10), the operation completes in constant time $O(1)$. In comparison, if we use a list structure to manage the nodes like in line 4 of Algorithm 2, the time complexity of searching the best N nodes becomes $O(n)$.

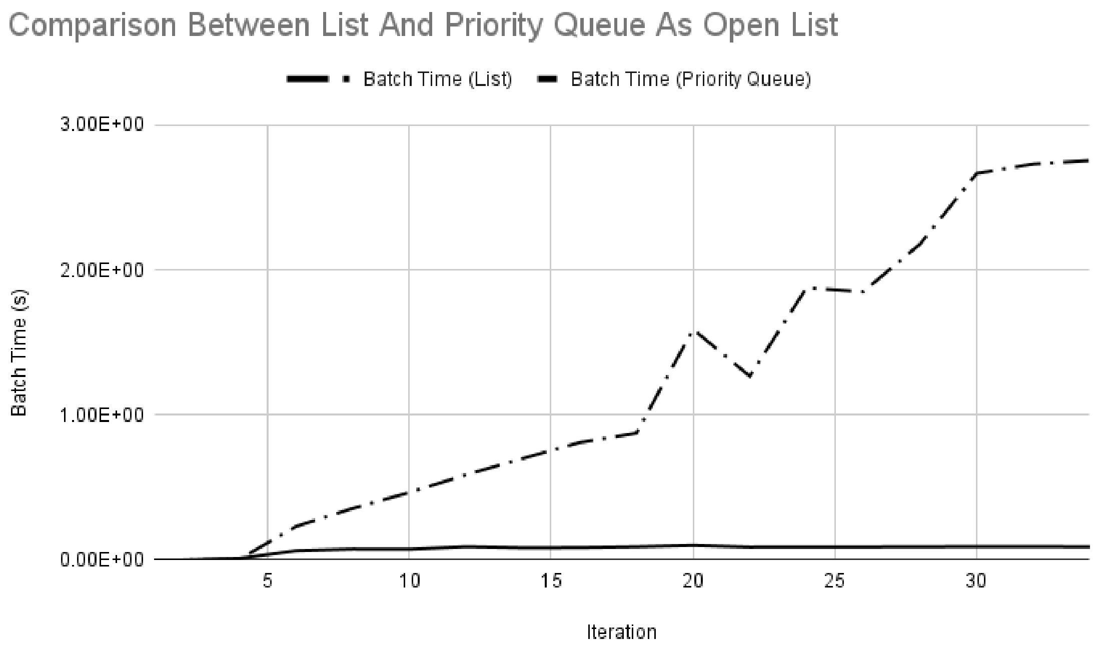


Figure 5.2: Comparison Between List And Priority Queue As Open List

As illustrated in Fig. 5.2, we compared the performance of using a priority queue versus a list to store nodes. The time required to assemble a batch of nodes for exploration in each iteration was measured. The results show that the priority queue consistently requires approximately the same amount of time to collect each batch which is around 0.1 seconds.

In contrast, using a list results in progressively increasing times as more nodes are added. For instance, by the 20th iteration, the open list contained $430,699$ nodes, and selecting the best N nodes took considerably longer due to the $O(n)$ complexity.

When adding nodes to the open list post-exploration, the priority queue manages this with a time complexity of $O(\log n)$, sorting nodes based on their f values. In contrast, inserting nodes into a list requires only $O(1)$ time. However, despite the logarithmic complexity, appending successors to a priority queue consistently takes approximately the same duration of about 0.2 seconds during the whole search in practice. This consistency is advantageous as the logarithmic operation does not scale with the number of nodes as steeply as an $O(n)$ operation would.

Given this efficiency, the priority queue is preferred for managing open nodes in both the Multi-Heuristic Batched Weighted A* search (Algorithm 10) and the Multi-Heuristic Anytime BWA* search (Algorithm 12). This choice optimizes the overall performance of our search strategies, balancing between time efficiency and computational scalability.

Enhanced Node Management with Dictionary Storage

To enhance the computational efficiency of our search algorithms, we employ a dictionary *CubeToSteps* to store the g values of nodes in both the open set and the closed list. This storage strategy increases memory usage but significantly speeds up the process of checking and adding new nodes to the open list, as described between lines 12 and 24 in Algorithm 2. When a successor node is considered for addition to the open list, it is evaluated not only against nodes in the closed list, if present, but also against any similar nodes in the open list. Using *CubeToSteps* consolidates operations from lines 12 to 24 into a more time-efficient sequence. The advantage of using a dictionary is its $O(1)$ time complexity for retrieving a value using a key, in contrast to the $O(n)$ time required to check for a state's presence in a list.

The dictionary stores and compares g values rather than f values. This choice is driven by the fact that the heuristic component (h value) remains constant between identical states, supported by the identical outputs from the deep model described in Chapter 4, which uses fixed parameters. Consequently, using g values alone suffices to evaluate and enhance the quality of potential solutions. We continuously update these g values in our dictionary: every time a node is removed from the open list (as noted in line 9 of Algorithm 10), and whenever we identify a superior g value for an existing state (referenced in line 33 of Algorithm 10).

This methodological improvement is similarly implemented in the Multi-Heuristic Anytime Weighted A* search (Algorithm 12), ensuring our algorithms are both resource-efficient and capable of handling complex search tasks effectively.

5.2.3 Full Implementation of MBWA* Search Algorithm

The full implementation can be seen in Algorithm 10.

The MBWA* search algorithm begins by initializing the *OpenList* as a priority queue that includes the initial node, as well as setting up the *CubeToSteps* dictionary as detailed in Section 5.2.2. The main execution loop starts at line 5, which continuously processes nodes until no more remain in the open list.

From lines 6 to 10, the algorithm selects the top B nodes from the *OpenList* and removes them. Line 11 involves analyzing the probability distribution of potential moves that could advance each of these top B nodes towards the solved state of the Rubik's cube, utilizing the model outlined in Section 4.4.

Subsequently, from line 12 to line 31, for each node among the top B, the process described in Section 4.5 is applied to assign the $g(n)$ and $f(n)$ values to each successor. Lines 32 to 38 involve checking each successor against the *CubeToSteps* dictionary to determine their potential relevance for future iterations.

If the *OpenList* is exhausted without finding a solution, the search is deemed unsuccessful. This structured approach ensures a systematic exploration of possible moves while efficiently prioritizing promising paths toward the solution.

Algorithm 10: MBWA* Search

```
   Data: $x_0$: Initially Scrambled Cube, $w$: Weights on $h(n)$, $B$: Batch Size
   Result: Solution or Failure.
 1 $OpenList \leftarrow PriorityQueue$;
 2 $f(x_0) = ComputeMinSteps(x_0), g(x_0) = 0$;
 3 $OpenList \leftarrow OpenList.HeapPush(f(x_0), x_0)$;
 4 $CubeToSteps \leftarrow Dictionary$;
 5 while $OpenList \neq \emptyset$ do
 6     $BestNodes \leftarrow$ The best $B$ node with the lowest $f(n) \in OpenList$;
 7     $OpenList \leftarrow OpenList \setminus \{BestNodes\}$;
 8     for $node \in BestNodes$ do
 9         $CubeToSteps[node] \leftarrow node.g$;
       /* Compute fitness with Algorithm 7 */
10     $p, h \leftarrow ComputeFitness(BestNodes)$;
11     for $node \in BestNodes$ do
12         for $successor \in node.successors$ do
13             if $successor$ is solved then
14                 if $|successor.moves| > 26$ then
15                     Continue;
16                 return $successor.moves$;
17             $h(successor) \leftarrow w \cdot (h[successor] - p[node][successor.moves[-1]])$;
18             $g(successor) \leftarrow g(node) + 1$;
19             $f(successor) \leftarrow g(successor) + h(successor)$;
20             if $g(successor) < CubeToSteps[successor]$ or
                 $successor.state \notin CubeToSteps$ then
21                 $CubeToSteps[successor] \leftarrow g(node)$;
22                 $OpenList \leftarrow OpenList \cup \{successor\}$;
23 return Search Failed;
```

5.2.4 Time Complexity of MBWA*

The time complexity of the classic A* search algorithm is generally given by $O(b^d)$, where b is the branching factor and d is the depth of the search [RN20]. In traditional A* search (Algorithm 2), the algorithm expands the most promising node at each step, considering all possible successors. Thus, the total number of nodes generated during this expansion provides a measure of the algorithm's time complexity:

$$N + 1 = 1 + b + b^2 + \cdots + b^d$$

However, the MBWA* incorporates a prevention algorithm as detailed in Section 3.2.3, specifically at line 14 in the MBWA* algorithm (Algorithm 10), which effectively reduces the branching factor. Traditionally, the branching factor for a Rubik's Cube is 12, accounting for all possible moves that can be applied. Yet, with the intervention of the prevention algorithm, the branching factor is reduced each time a node is expanded from the current root (excluding the initial depth). Depending on the specific moves already made and outlined in Table 3.2 within Section 3.2.3, the branching factor can decrease to 11 or even as low as 8.

This reduction in the branching factor leads to a decreased time complexity for the MBWA* search compared to the classic A* search. By efficiently lowering the number of potential paths explored at each step, MBWA* not only expedites the search process but also significantly reduces computational overhead, making it a more efficient option for solving complex problems such as the Rubik's Cube.

5.3 Multi-heuristic Beam Search Algorithm (MBS)

We have developed an enhanced version of the beam search algorithm, referred to as the Multi-heuristic Beam Search (MBS). Beam search was chosen for modification due to its inherent simplicity and fast execution, facilitated by a predefined limit on the number of nodes explored. For this implementation, we limit our search to a maximum depth of 26 moves, effectively constraining the solution space.

5.3.1 Utilizing a Seen-State Set

In the standard beam search, outlined in Algorithm 1, each generation's successors are added to the candidate pool for the subsequent generation. This approach can lead to redundant effort and increased resource usage due to the re-exploration of previously searched states. Given that a particular cube state can be achieved through different sequences of moves, it's essential to track which states have been visited. For instance, as illustrated in Fig. 3.15, the same cube state can be reached by the move sequence M_e as well as an alternative sequence $M'_e =$ `L L U' L' U' L U L U L U' L`. Both sequences result in the same final state but follow different paths.

To mitigate the inefficiency caused by revisiting the same states, we incorporate a seen-state set. This set records each state that has been explored, allowing the algorithm to quickly ascertain whether a state has already been searched. This strategy significantly reduces both the time spent searching and the memory required, as it avoids unnecessary duplication of effort.

Importantly, we utilize a set rather than a dictionary for tracking seen states because the f value of nodes remains constant throughout the beam search once it is computed, which progresses layer by layer without consideration of path cost (g value). Therefore, a simple set is adequate for our needs, streamlining the algorithm and enhancing its performance by focusing solely on state uniqueness rather than path metrics.

5.3.2 Full Implementation of MBS Algorithm

The full implementation can be seen in Algorithm 11.

The MBS search algorithm initiates by setting up the *Beam* as a list that contains

Algorithm 11: Multi-heuristic Beam Search

```
Data: $x_0$: Initially Scrambled Cube, $BeamSize$
Result: Solution or Failure.
1  $Beam \leftarrow \{x_0\}$;
2  $SeenStates \leftarrow \{x_0.cube\}$;
3  $depth = 0$;
4  while $Beam \neq \emptyset$ and $depth \leq 26$ do
       /* Compute fitness of successors with Algorithm 7 */
5      $p, h \leftarrow ComputeFitness(Beam)$;
6      $NextBeam \leftarrow \{\ \}$;
7      for $node \in Beam$ do
8          for $successor \in node.successors$ do
9              if $successor$ is solved then
10                 return $successor.moves$;
11             $f(successor) \leftarrow h[successor] - p[node][successor.moves[-1]]$;
12             $SeenStates \leftarrow SeenStates \cup successor$;
13     $Beam \leftarrow BeamSize$ nodes with lowest $f(n)$ from $NextBeam$;
14     $depth \leftarrow depth + 1$;
15 return Search Failed;
```

the initial node. Simultaneously, it establishes the *SeenState* set with the initial cube state included, as outlined in Section 5.3.1. The main execution loop commences at line 4, continuously processing nodes until the *Beam* is empty or the maximum search depth is reached.

Within the loop, the algorithm processes each node in the *Beam*, selecting the best *BeamSize* nodes as candidates in a step-by-step manner. From lines 5 to 26, each node within the *Beam* is expanded, and the process described in Section 4.5 is utilized to assign $f(n)$ values to each successor. Lines 27 to 29 involve selecting the top *BeamSize* nodes from the successors generated at the previous depth.

As detailed in Section 2.3, the nodes in the *Beam* list at depth i represent the most promising nodes, having undergone i moves from the scrambled state. If no solution is produced by depth 27, this indicates that all potential nodes with a move sequence length of 26 have been exhaustively searched, and thus the search is concluded as unsuccessful.

5.3.3 Time Complexity of MBS

Assuming a beam size of b, a branching factor of m, and a maximum search depth of d, the maximum number of nodes searched in MBS can be calculated as $d \cdot m \cdot b$. At each depth, the nodes within the beam list, which contains b nodes, are expanded with a branching factor of m. Given that the search can go as deep as d levels, the total number of nodes explored is capped at $d \cdot m \cdot b$. This formula provides a clear upper bound on the computational effort required by the MBS algorithm, demonstrating its scalability and efficiency relative to the parameters set for beam size, branching factor, and search depth.

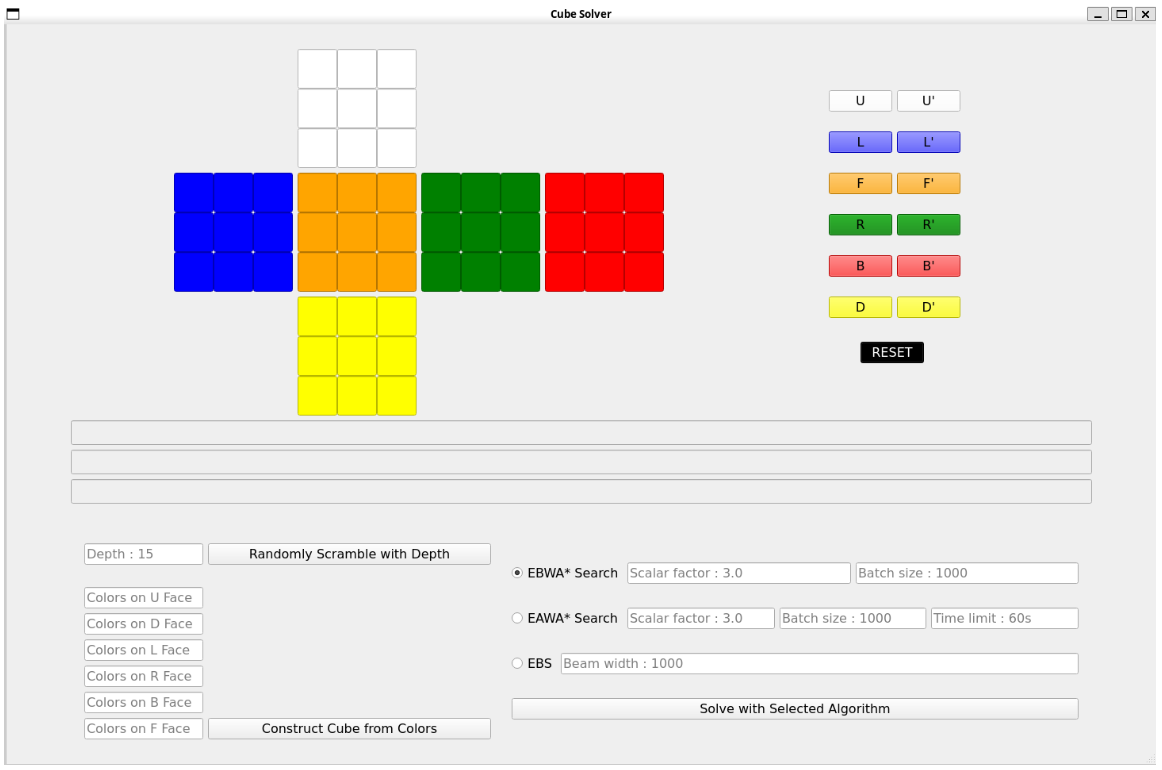
5.4 Anytime Variation of MBWA*

This section will focus on the third goal mentioned as we want to improve the solution quality (the fewest move count to solve the cube) within a given time frame. We modify MBWA* to make it an anytime algorithm following the idea of Eric A. Hansen et al. [HZ07]

The main difference between MAWA* and MBWA* is the addition of the incumbent attributes to the algorithm. If we have found a solution before, then each time when we want to find a node to explore or find another solution, instead of comparing the weighted fitness $f'(n) = g(n) + w \cdot (n)$, we compare the direct fitness $f(n) = g(n) + h(n)$ which put more focus on g that can improve the solution quality.

5.5 Complete Solver

The complete solver integrates all the algorithms previously discussed, utilizing PyQt5 [Riv23] for the graphical user interface, inspired by the user interface created by [Bil22]. An illustration of the solver's user interface is provided in Fig. 5.3.


Cube Solver
U
U'
L
L'
F
F'
R
R'
B
B'
D
D'
RESET
Randomly Scramble with Depth
Construct Cube from Colors
Solve with Selected Algorithm

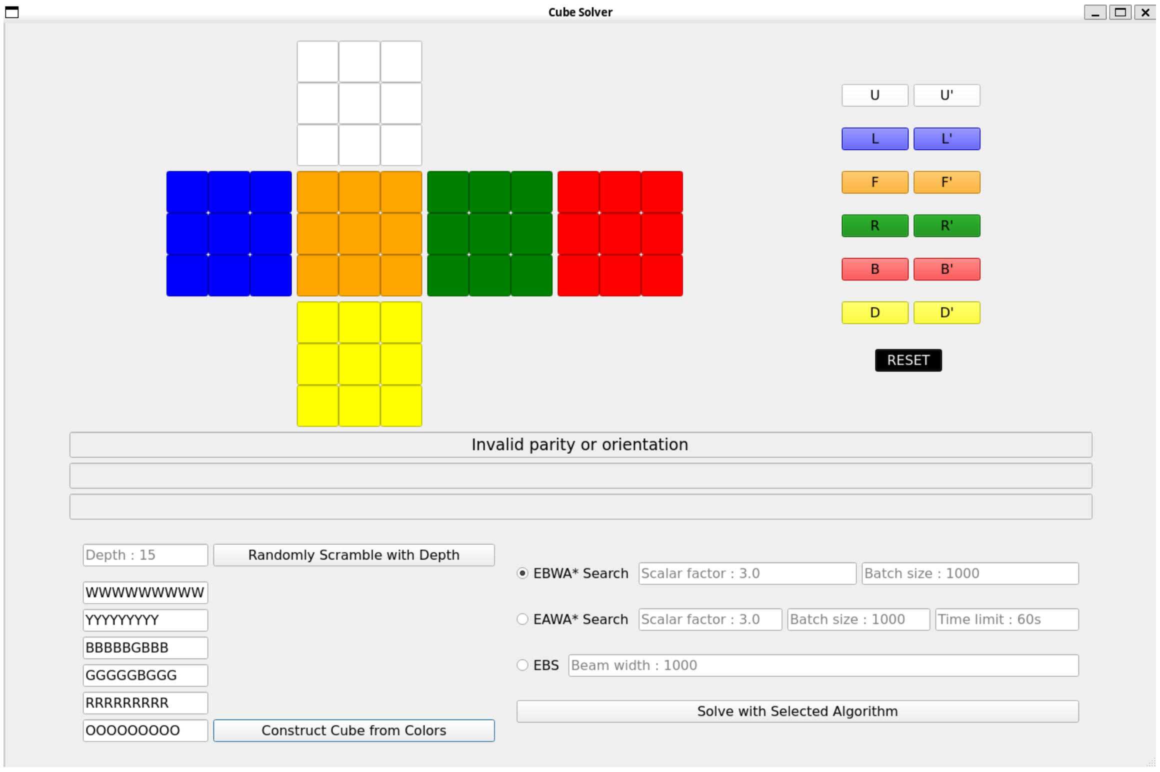
Depth : 15

Colors on U Face
Colors on D Face
Colors on L Face
Colors on R Face
Colors on B Face
Colors on F Face

[✓ EBWA* Search] Scalar factor : 3.0 Batch size : 1000
EAWA* Search Scalar factor : 3.0 Batch size : 1000 Time limit : 60s
EBS Beam width : 1000

Figure 5.3: User Interface of the Complete Cube Solver

Within this solver, users have the option to randomly scramble the cube to a user-defined depth or input their own cube configurations by inputting the facet-level cube string. Each configuration is validated against Algorithm 9. If the configuration fails validation, an error message is displayed, prompting the user to revise their input. For instance, entering an invalid configuration as shown in Fig. 3.1 triggers an error message displayed by the solver, as illustrated in Fig. 5.4.


Cube Solver
U
U'
L
L'
F
F'
R
R'
B
B'
D
D'
RESET
Randomly Scramble with Depth
Construct Cube from Colors
Solve with Selected Algorithm

Invalid parity or orientation

Depth : 15

WWWWWWWWWW
YYYYYYYYY
BBBBBGBBB
GGGGGBGGG
RRRRRRRRR
OOOOOOOOO

[✓ EBWA* Search] Scalar factor : 3.0 Batch size : 1000

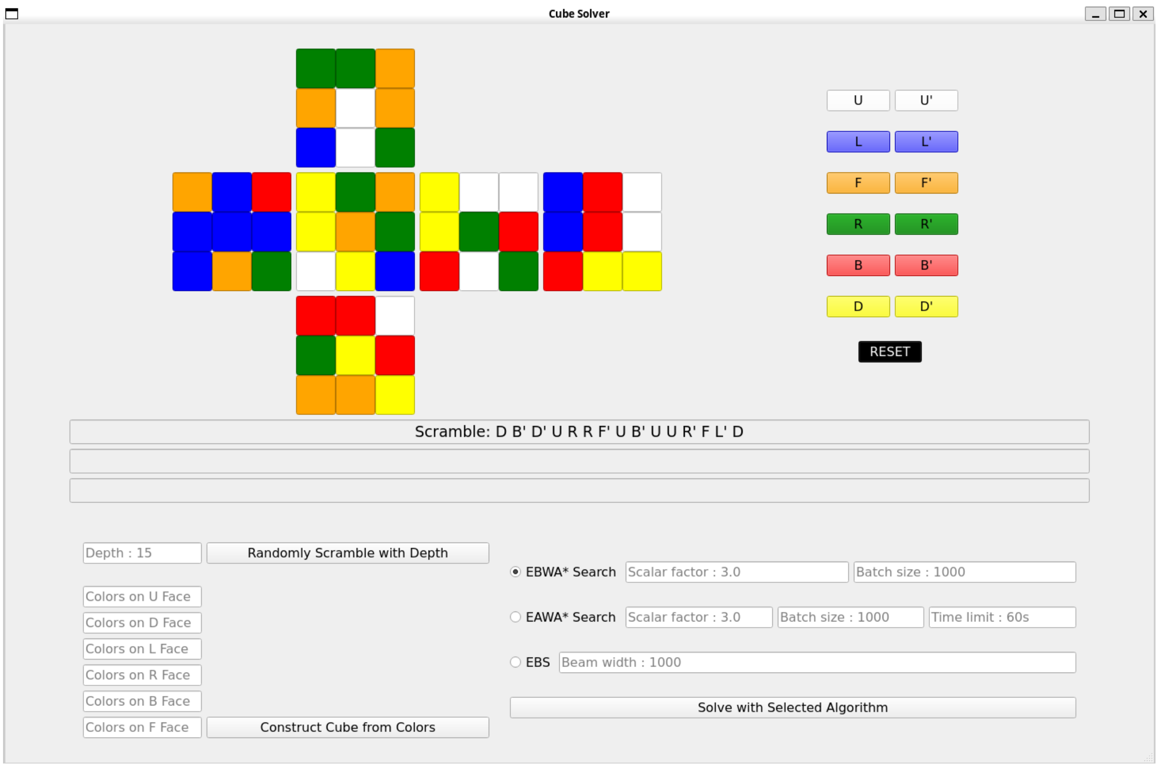
EAWA* Search Scalar factor : 3.0 Batch size : 1000 Time limit : 60s

EBS Beam width : 1000

Figure 5.4: Cube Solver With The Invalid State

Users can select from various search algorithms and adjust parameters according to their specific objectives, located in the right bottom corner of the interface. Post-solve, the solver displays statistics such as the solution sequence, solving time, and more, as depicted in Fig. 5.5 and Fig. 5.6.

This solver offers a user-friendly platform for interacting with the various implemented algorithms, including MBWA*, MAWA*, and MBS, providing a practical demonstration of their capabilities. This concludes the discussion on the implementation of the solver.



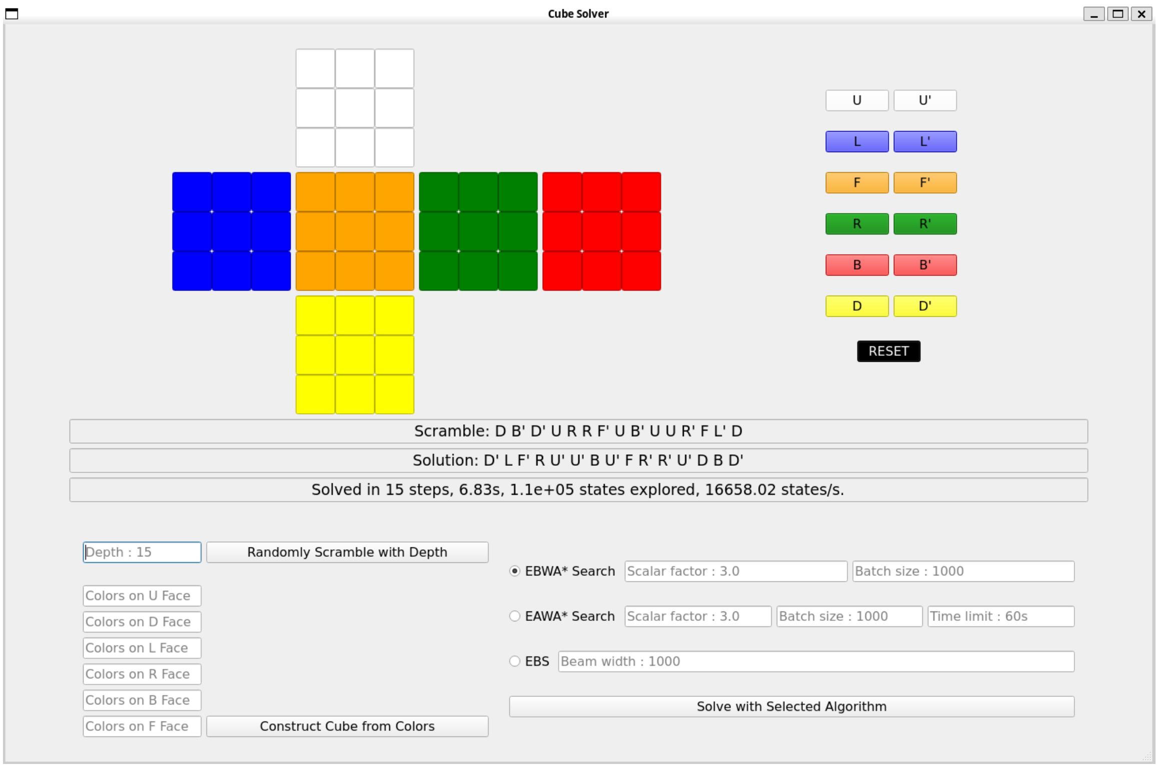
Scramble: D B' D' U R R F' U B' U U R' F L' D

Depth : 15 | Randomly Scramble with Depth

Colors on U Face
Colors on D Face
Colors on L Face
Colors on R Face
Colors on B Face
Colors on F Face | Construct Cube from Colors

[✓ EBWA* Search] Scalar factor : 3.0 | Batch size : 1000
EAWA* Search Scalar factor : 3.0 | Batch size : 1000 | Time limit : 60s
EBS Beam width : 1000
Solve with Selected Algorithm

Figure 5.5: Cube Solver With Scrambled Cube



Scramble: D B' D' U R R F' U B' U U R' F L' D

Solution: D' L F' R U' U' B U' F R' R' U' D B D'

Solved in 15 steps, 6.83s, 1.1e+05 states explored, 16658.02 states/s.

Depth : 15 | Randomly Scramble with Depth

Colors on U Face
Colors on D Face
Colors on L Face
Colors on R Face
Colors on B Face
Colors on F Face | Construct Cube from Colors

[✓ EBWA* Search] Scalar factor : 3.0 | Batch size : 1000
EAWA* Search Scalar factor : 3.0 | Batch size : 1000 | Time limit : 60s
EBS Beam width : 1000
Solve with Selected Algorithm

Figure 5.6: Cube Solver With The Solution Statistics

Algorithm 12: MAWA* Search

```
Data: $x_0$: Initially Scrambled Cube, $w$: Weights on $h(n)$, $B$: Batch Size,
      $MaxRunningTime$
Result: Solution or Failure.
1  $StartTime \leftarrow CurrentTime$;
2  $OpenList \leftarrow PriorityQueue$;
3  $f(x_0) = ComputeMinSteps(x_0), f'(x_0) = w \cdot f(x_0), g(x_0) = 0$;
4  $OpenList \leftarrow OpenList.push(f'(x_0), x_0)$;
5  $CubeToSteps \leftarrow Dictionary$;
6  $Incumbent \leftarrow None$;
7  while $OpenList \neq \emptyset$ and $CurrentTime - StartTime \leq MaxRunningTime$
    do
8      $BestNodes \leftarrow \{ \}$;
9      while $|BestNodes| \leq B$ and $OpenList$ do
10         $node \leftarrow OpenList.pop()$;
11         $CubeToSteps[node] \leftarrow node.g$;
12         if $incumbent == None$ or $node.f < incumbent.f$ then
13             $BestNode \leftarrow BestNode \cup \{BestNode\}$;
14     $p, h \leftarrow ComputeFitness(BestNodes)$;
15     for $node \in BestNodes$ do
16         for $successor \in node.successors$ do
17             $h(successor) \leftarrow (s[successor] - p[node][successor.moves[-1]])$;
18             $g(successor) \leftarrow g(node) + 1$;
19             $f(successor) \leftarrow g(successor) + h(successor)$;
20             $f'(successor) \leftarrow g(successor) + w \cdot h(successor)$;
21             if $successor$ is solved then
22                 if $incumbent$ is $None$ and or $f(successor) < f(incumbent)$
                    then
23                     $incumbent \leftarrow successor$
24             if $g(successor) < CubeToSteps[successor]$ or
                $successor.state \notin CubeToSteps$ then
25                 $CubeToSteps[successor] \leftarrow g(node)$;
26                 $OpenList \leftarrow OpenList \cup \{successor\}$;
27 if $incumbent \neq None$ then
28     return $incumbent.moves$;
29 return Search Failed;
```

Chapter 6

Experimental Results and Interpretation

In this chapter, we evaluate the effectiveness of the improvements made to the algorithms discussed in the previous chapter. We focus on five specific scrambles that were used in the World Cube Association Championship, where Yiheng Wang achieved a world-record average solve time of 4.48 seconds in Singapore [Wor24]. These scrambles are used as the initial benchmark to test the enhanced algorithms. Subsequent testing will extend to a broader set of 100 random scrambles from the DeepCubeA dataset [AMSB19], providing a more comprehensive evaluation. All experiments were conducted using an Nvidia RTX 3090 graphics card, ensuring consistency across tests.

6.1 MBWA* Compared With A* Search

We begin by evaluating the performance of Batched Weighted A* (BWA*) in comparison to Weighted A* (WA*). As discussed in Section Section 5.2.1, BWA* was anticipated to be more efficient in terms of time, expand more nodes than WA*, and deliver comparable solution quality.

The results, as detailed in Table Table 6.1, confirm these expectations. The success rate (SR) indicates the proportion of successfully solved problems, while Time (s) represents the average solve time across five attempts. The average solution length ($|Sol|$) and the number of nodes expanded (N ($\times 10^3$)) are also reported for each algorithm. In every trial, BWA* consistently outperforms WA* by taking less time—often less than half—and expanding more nodes, yet achieving a similar solution quality.

This superior performance of BWA* is due to its ability to expand multiple nodes per iteration, thereby increasing its chances of finding a solution more quickly. Leveraging the batch processing capabilities of the Nvidia RTX 3090 and models, as detailed in Section Section 2.4.2, BWA* can compute the distances for multiple nodes almost as fast as it does for a single node. This efficiency significantly reduces the overall computation time.

Furthermore, we evaluate the performance of Multi-Heuristic Batched Weighted A* (MBWA*), which incorporates a multi-heuristic function as detailed in Section

Table 6.1: Comparison of WA* and BWA* with Batch Size of 500

w	**WA***				**BWA*** ($B = 500$)			
	SR	**Time (s)**	$\|Sol\|$	**N** ($\times 10^3$)	**SR**	**Time (s)**	$\|Sol\|$	**N** ($\times 10^3$)
2.4	1.0	159.11	23.8	1060.16	1.0	46.01	23.8	1168.93
2.6	1.0	67.63	23.8	448.58	1.0	19.67	23.8	497.15
2.8	1.0	27.79	24.2	188.09	1.0	13.26	23.4	341.43
3.0	1.0	26.45	23.8	175.05	1.0	12.82	24.2	322.75

Section 4.5, against BWA* that employs a standard deep reinforcement learning model from Section 4.3. For this comparative analysis, we utilize scalar weights of 3 and adjust the batch size B.

From the data presented in Table Table 6.2, it's evident that Multi-Heuristic Batched Weighted A* (MBWA*) performs more efficiently than BWA* across all metrics. For instance, with a batch size of 70, MBWA* achieves an average solve time of 1.97 seconds compared to 13.08 seconds for BWA*, representing a 90.44% increase in solving speed. Although MBWA* tends to yield slightly longer solutions—due to placing a greater emphasis on the heuristic component (h)—this trade-off is justified by its primary objective: to find solutions as quickly as possible while keeping the solution length below 26 moves.

Additionally, MBWA* exhibits greater stability compared to BWA*. As batch size increases, MBWA*'s average search time and node expansion show a consistent gradual increase. In contrast, when B is 200, BWA*'s shows an abrupt decrease in search time followed by a rebound. This instability in BWA* can be attributed to the use of an inadmissible heuristic function derived from the deep reinforcement learning model discussed in Section Section 4.5.

The advantage of MBWA* is largely attributed to the integration of two models that guide the heuristic search, as detailed in Section Section 4.5. Both models suffer from a deep scramble depth, but when the scramble depth is around 10 and below 10, they tend to perform better and make accurate predictions. By emphasizing the heuristic component (h) and ensuring it is both admissible and consistent, the search process not only accelerates but also maintains stability. This strategic approach enhances the performance of MBWA*, enabling it to achieve faster solve times while maintaining reliability in the results.

6.2 MBS compared with Classic Beam Search

One major enhancement discussed is the incorporation of the *SeenState* set, as detailed in Section Section 5.3.1. We evaluate the impact of this modification by comparing the performance of beam search with and without the *SeenState* set, as shown in Table Table 6.3. Note that the success rate is not 1.0 for some trials due to the incompleteness of beam search discussed in Section 2.3. Therefore, success rate becomes a crucial metric for evaluating the performance of different beam search variations.

The results indicate that beam search with *SeenState* (BSwSS) either matches or

Table 6.2: Comparison of BWA* and MBWA* with Scale Factor 3

B	**BWA***				**MBWA***			
	SR	**Time (s)**	$\|Sol\|$	**N** ($\times 10^3$)	**SR**	**Time (s)**	$\|Sol\|$	**N** ($\times 10^3$)
70	1.0	13.08	24.2	314.76	1.0	1.97	25.0	46.86
100	1.0	13.87	24.2	345.26	1.0	2.31	25.0	56.72
200	1.0	9.97	24.2	251.80	1.0	2.50	24.6	64.05
500	1.0	12.72	24.2	322.75	1.0	4.27	24.6	110.30
700	1.0	13.00	24.2	330.44	1.0	5.99	24.6	152.22
1000	1.0	15.90	24.2	403.43	1.0	8.56	24.2	214.44
2000	1.0	19.50	24.2	492.59	1.0	15.61	23.4	393.69

exceeds the success rate of beam search without *SeenState* (BSw/oSS). This improvement suggests that by incorporating *SeenState* and minimizing redundant searches, BSwSS achieves a higher success rate in solving the cube.

Despite this enhanced success rate, it is noteworthy that for the states that are solved, BSw/oSS completes them in less time. Additionally, BSwSS tends to expand fewer nodes compared to BSw/oSS, thanks to its effective reduction of duplicated state explorations. This balance between efficiency and efficacy highlights the practical benefits of the *SeenState* set in optimizing search strategies and enhancing overall performance.

Table 6.3: Comparison of BeamSearchWithoutSS and BeamSearchWithSS

B	**BSw/oSS**				**BSwSS**			
	SR	**Time (s)**	$\|Sol\|$	**N** ($\times 10^3$)	**SR**	**Time (s)**	$\|Sol\|$	**N** ($\times 10^3$)
300	0.4	1.98	25.0	68.775	0.4	2.50	25.0	65.565
500	0.4	3.31	25.0	113.385	0.6	4.07	24.67	106.762
1000	0.4	6.56	25.0	225.461	1.0	8.38	25.0	215.583
1500	0.4	9.80	25.0	336.537	1.0	12.54	25.0	318.317
2000	0.4	12.88	25.0	443.250	1.0	16.35	24.6	412.683
2500	0.8	16.37	25.25	557.220	1.0	20.46	24.6	512.930
3000	1.0	19.36	25.0	657.368	1.0	24.06	24.2	601.064
5000	1.0	31.86	24.2	1041.457	1.0	40.41	24.2	993.571

Building on the analytical approach from the previous section, we extend our comparison to Multi-Heuristic Beam Search (MBS) and Beam Search with *SeenState* (BSwSS), using the results from Table Table 6.4. The implementation of a multi-heuristic approach in MBS significantly improves the success rates for beam sizes (B) of 300 and 500, boosting them from 0.4 to 0.8 and 0.6 to 1.0, respectively.

Moreover, MBS not only achieves higher success rates but also requires less time to find solutions and expands fewer nodes compared to BSwSS. This increased efficiency is complemented by the ability of MBS to find better solutions, attributed to the guidance provided by the multi-heuristic. This further validates the effectiveness of the multi-heuristic approach shown in Section 4.5 in optimizing search algorithms.

Table 6.4: Comparison of BeamSearchWithSS and MBS

B	**BSwSS**				**MBS**			
	SR	**Time (s)**	$\|Sol\|$	**N** ($\times 10^3$)	**SR**	**Time (s)**	$\|Sol\|$	**N** ($\times 10^3$)
300	0.4	2.50	25.0	65.565	0.8	2.50	24.5	64.830
500	0.6	4.07	24.67	106.762	1.0	4.13	24.2	105.648
1000	1.0	8.38	25.0	215.583	1.0	8.23	24.2	209.525
1500	1.0	12.54	25.0	318.317	1.0	11.77	23.4	297.305
2000	1.0	16.35	24.6	412.683	1.0	15.55	23.4	392.239
2500	1.0	20.46	24.6	512.930	1.0	19.45	23.4	487.422
3000	1.0	24.06	24.2	601.064	1.0	22.85	23.0	570.772
5000	1.0	40.41	24.2	993.571	1.0	38.39	23.0	942.683

6.3 Minimal Solving Time with MBWA* and MBS

As shown in the previous two sections, both MBS and MBWA* have demonstrated superior performance compared to their respective original search algorithms. This section aims to identify the optimal algorithms and parameters for achieving the fastest possible solution times, using the same Nvidia RTX 3090 platform for all tests.

Shown in Table 6.5, the quickest average solution time recorded is 1.97 seconds, achieved by MBWA* using a scalar factor of 3 and a batch size of 70. Although MBS has shown the capability to achieve sub-one-second average solving times, its success rate does not reach 100%. Therefore, when prioritizing reliability alongside speed, the choice falls on MBWA* at the mentioned settings, which consistently delivers a 100% success rate with an average time of 1.97 seconds. This emphasizes the effectiveness of MBWA* in balancing rapid performance with complete reliability in solving outcomes.

Table 6.5: Comparison of MBS and MBWA* Search

B	**MBS**				**MBWA***			
	SR	**Time (s)**	$\|Sol\|$	**N** ($\times 10^3$)	**SR**	**Time (s)**	$\|Sol\|$	**N** ($\times 10^3$)
50	0.2	0.58	26.0	12.111	1.0	4.56	25.0	102.389
70	0.2	0.81	26.0	16.925	1.0	1.97	25.0	46.864
100	0.2	0.97	26.0	24.054	1.0	2.39	25.0	56.716
300	0.8	2.57	24.5	64.830	1.0	3.30	25.0	83.593
320	1.0	2.72	24.6	69.405	1.0	3.31	25.0	83.074
500	1.0	4.13	24.2	105.648	1.0	4.28	24.6	110.301

6.4 MBWA* Compared with MAWA*

This section delves into the “anytime” adaptation of the MBWA* search, termed MAWA* (Multi-Heuristic Anytime Weighted A*), which aims to enhance solution quality within a constrained timeframe of 60 seconds.

Table 6.6: Comparison of MBWA* and MAWA* with Scale Factor 3

Batch Size	**MBWA***				**MAWA***			
	SR	**Time (s)**	$\|Sol\|$	**N** $(\times 10^3)$	**SR**	**Time (s)**	$\|Sol\|$	**N** $(\times 10^3)$
70	1.0	1.97	25.0	46.86	1.0	60.45	22.6	1254.59
100	1.0	2.31	25.0	56.72	1.0	60.43	22.6	1299.92
200	1.0	2.50	24.6	64.05	1.0	60.31	22.6	1326.46
500	1.0	4.27	24.6	110.30	1.0	60.36	22.6	1343.16
700	1.0	5.99	24.6	152.21	1.0	60.50	22.6	1350.63
1000	1.0	8.56	24.2	214.44	1.0	60.39	22.6	1342.02
2000	1.0	15.61	23.4	393.69	1.0	60.88	22.6	1368.22

Data from Table Table 6.6 reveals that MAWA* consistently achieves superior solution lengths across all tested batch sizes, with an average solution length of 22.6. This marks a significant improvement over MBWA*, which achieves its best average solution length of 23.4, indicating that MAWA* typically finds solutions that are approximately one move shorter.

The success of MAWA* in improving solution quality can be due to its strategy of evaluating potential solutions based on their original f values rather than a weighted score. This approach allows MAWA* to prioritize paths that may lead to more optimal solutions, leveraging additional processing time to explore these paths more thoroughly. As a result, MAWA* effectively utilizes the allotted time to enhance the quality of solutions, demonstrating its efficacy as an "anytime" algorithm capable of dynamically refining its search to yield better outcomes as time permits.

6.5 Comprehensive Results

The previous sections were based on the average scores from five scrambles. In this section, we will test MBWA* (Multi-Heuristic Batched Weighted A*), MBS (Multi-Heuristic Beam Search), and MAWA* (Multi-Heuristic Anytime Weighted A*) using 100 random scrambles from the dataset cited in [AMSB19]. All the tables and data from these tests are included in Appendix C and are not presented in this section.

6.5.1 MBWA* and MBS

Based on the average solving times (Table C.2), average solution lengths (Table C.1), and average nodes expanded (Table C.3) for MBWA* and MBS, we visualize these results in Fig. 6.1 and Fig. 6.2.

The data indicate that solving times for MBWA* tend to increase gradually as the batch size increases and the scalar factors decrease, correlating with an increase in node expansion. As scalar factors increase further, the rate of decrease in solving time diminishes, likely approaching the processing speed's limit and theoretical lower bounds of solving efficiency.

Particularly, consider MBWA* with a scalar factor of 8 and a batch size of 500 and 700, the average solving time reaches above 200 seconds. This suggests that the MBWA* searches with low batch sizes and high scalar factors encounter difficulty in finding a solution, likely due to residual inadmissibility in the multi-heuristic function.

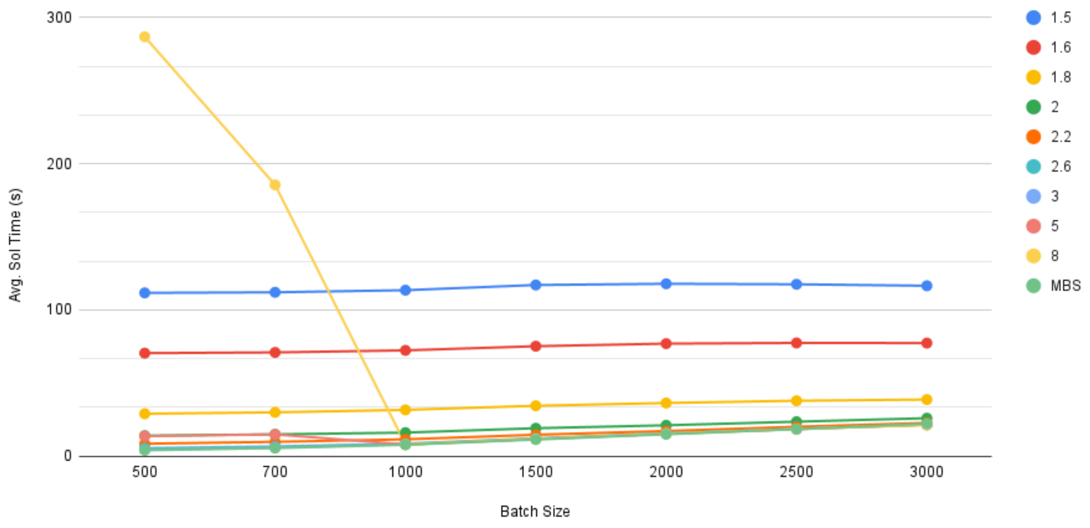


Figure 6.1: Comparative Analysis of Average Solving Time by MBWA* and MBS

Furthermore, we explore the average solution length across different parameter settings of MBWA* and MBS. The data reveal a clear trend: lower scalar weights and higher batch sizes generally lead to better solution quality. Additionally, as the scalar factor increases, increasing the batch size appears to have a more pronounced positive effect on solution quality.

Considering the inherent nature of MBS, which solely utilizes the heuristic function, it operates comparably to MBWA* when high scalar factors are employed. Essentially, MBS functions as if an indefinitely high scalar factor is applied to the heuristic component (h), making its operational dynamics similar to those of MBWA* with scalar factors set to values like 5.0 or 8.0.

It is important to note that the success rates for MBS with beam sizes of 500 and 700 are 0.97 and 0.98, respectively. These figures reflect the inherent limitations of beam search, where the completeness of the search is not guaranteed. The inadmissibility of the heuristic function can lead to potential solutions being overlooked if the heuristic inaccurately estimates the costs at certain depths, which explains the slight shortfall in achieving perfect success rates. This underlines the critical impact of heuristic accuracy on the search's effectiveness, especially in depth-level estimations.

For achieving the fastest possible solves with optimal efficiency, our comprehensive testing indicates that the MBWA* search algorithm performs exceptionally well. The MBWA* search, when configured with batch sizes ranging from 200 to 1000 and scalar factors between 2.6 and 3.4, demonstrates stable and efficient performance, typically completing within a time frame of 3 to 10 seconds. This specific range of parameters effectively balances performance with computational efficiency, making it a preferred choice for scenarios that require rapid and reliable solutions.

6.5.2 MAWA* Compared With MBWA*

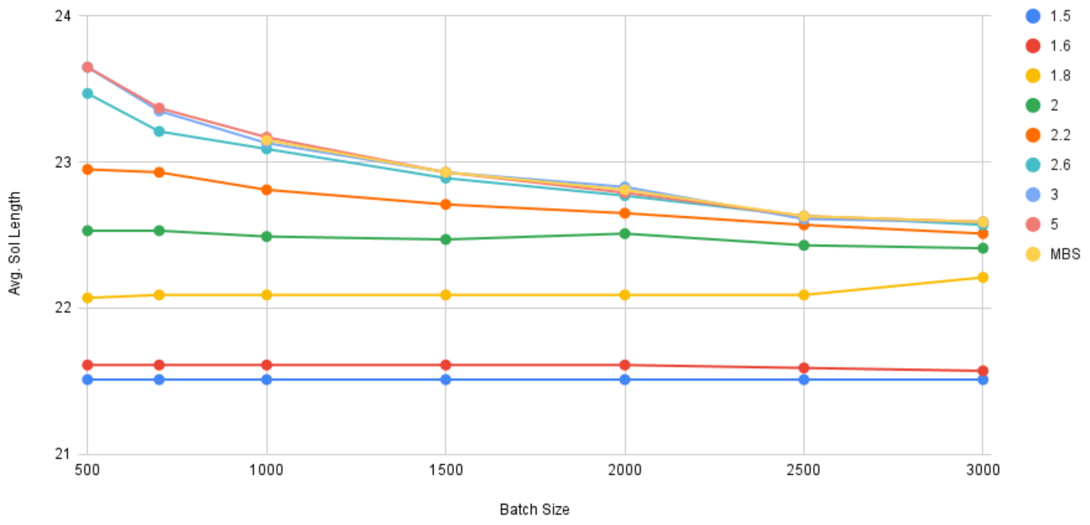


Figure 6.2: Comparative Analysis of Average Solution Lengths by MBWA* and MBS

In this section, we analyze the solution lengths generated by MAWA* in comparison with those from MBWA*. The findings, illustrated in Fig. 6.3, indicate that as the batch size increases, the degree of improvement in solution length diminishes. Specifically, for scalar factors of 1.6, 1.8, and 2.0, the improvements stabilize around an average of 0.5 moves shorter.

However, it's important to note that the success rate of MAWA* when the scalar factor is below 2.2 is not 100%. This indicates that some scrambles do not find a solution within 60 seconds, as detailed in Table C.5. Consequently, focusing solely on scalar factors of 2.2, 2.6, and 3.0, we observe that the improvement in solution length decreases as the batch size increases. The range of improvements in solution quality varies from 0.4 to 1.3 moves, which corresponds to approximately 5% of the overall solution length. This suggests that while MAWA* can enhance solution quality, the extent of improvement is influenced significantly by the chosen parameters, particularly the scalar factor and batch size.

It appears that the best achievable result for MAWA* over the 100 scrambles is an average solution length of approximately 22 steps. This is about 0.7 steps longer than the theoretical optimal solution length. Given the vast search space of the Rubik's Cube, calculating the optimal solution within a 60-second timeframe is impractical. Nonetheless, MAWA* clearly demonstrates its potential for enhancing solution quality when compared to MBWA*.

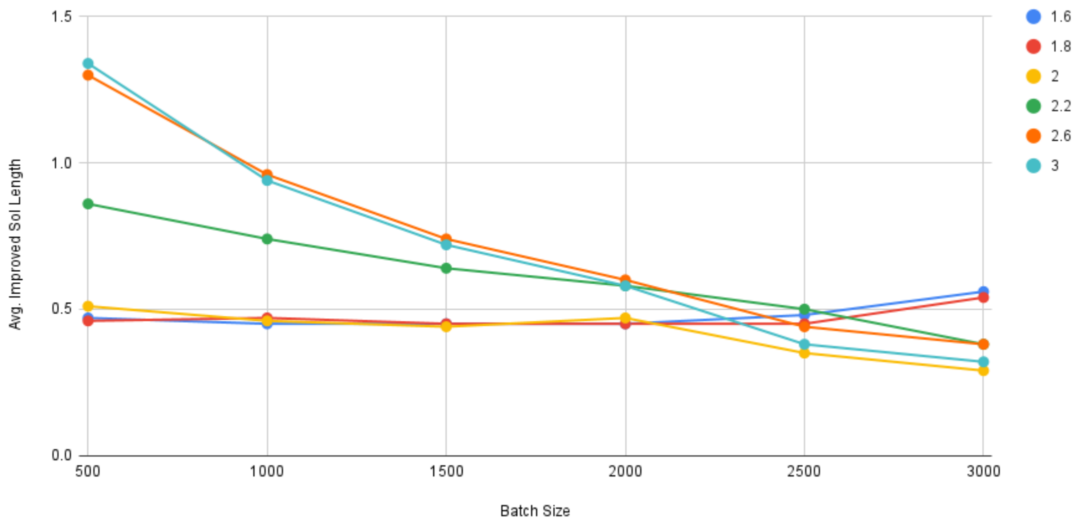


Figure 6.3: Comparative Analysis of Average Improved Solution Lengths by MBWA* and MAWA*

Chapter 7

Conclusion and Discussion

7.1 Summary of Findings

This research successfully demonstrated the application and effectiveness of various heuristic search algorithms combined with deep learning techniques and mathematical properties to solve the Rubik's Cube efficiently. The Multi-heuristic Batched Weighted A* (MBWA*) and Multi-heuristic Beam Search (MBS) algorithms, enhanced by deep reinforcement learning, showed significant improvements in solving time and solution optimality compared to traditional methods like A* and Greedy Best-First Search.

In this thesis, we explore the Rubik's Cube group and prove the first law of cubology innovatively which builds up the foundation of understanding the structure of the Rubik's Cube. This thesis provides a novel combination of the two deep learning models together to create a new heuristic function to guide the search for the solution of the Rubik's cube. Combined with group theory, we successfully designed the algorithm to check the validity of a Rubik's cube state and reduce the branching factor of search algorithms to make them even faster and more accurate.

The findings suggest that integrating deep learning models with heuristic searches provides a robust framework for tackling complex problems like solving the Rubik's Cube where traditional algorithms falter due to the vast solution space or computational constraints. The MBWA* algorithm, in particular, proved adept at managing the balance between search depth and breadth, showcasing fewer backtrackings and higher accuracy in predicting the optimal solution path.

7.2 Strengths and Weaknesses of Each Approach

The MBWA* algorithm excels in its capacity to dynamically adjust search parameters based on the learned heuristic, which significantly decreases solution times for complex Rubik's Cube states. It offers flexibility in tuning, such as adjusting the scalar factor to balance speed and solution quality. However, this algorithm depends heavily on a high-quality training dataset and requires substantial initial training time. Additionally, it demands more memory space compared to the MBS algorithm, which uses a fixed beam size to limit node storage.

The MBS algorithm achieves faster initial results but tends to struggle with achieving optimal solutions, indicating a need for heuristic refinement. Its primary limitation stems from its design; the fixed beam size introduces a degree of incompleteness, making it impossible to avoid trade-offs between memory usage and computational time. This is a manifestation of the ”No Free Lunch” theorem, where improvements in one area typically come at the cost of performance in another.

As for the MAWA* algorithm, it marginally improves solution quality but is constrained by the time limits set and the challenges of searching for better solutions with an inadmissible heuristic. This inherent limitation highlights the delicate balance between computational feasibility and the pursuit of higher solution accuracy.

7.3 Future Research Direction

Future studies should consider extending the current model to tackle the 4×4 Rubik's Cube, which presents a significantly greater complexity due to the additional elements like shifting center cubies. This complexity necessitates deeper analytical approaches and novel algorithms tailored to the enhanced challenge.

Furthermore, the learning architecture used in solving the Rubik's Cube could benefit from advancements in machine learning technologies. As highlighted in Chapter 4, the performance of existing models diminishes with increased scramble lengths. Employing more sophisticated models, such as convolutional neural networks or alternative architectures, could potentially enhance the accuracy of feature estimations, particularly in fully scrambled states.

Adjustments to the training process also present a promising avenue for research. For instance, altering the distribution of training data to include a higher proportion of highly scrambled states could improve model robustness. Additionally, experimenting with the discount factor γ, especially setting $\gamma < 1$ in the Deep Reinforcement Learning (DRL) model, might yield interesting outcomes in terms of learning efficacy and algorithm performance.

Moreover, the exploration of more advanced heuristic functions could lead to more efficient search algorithms. Techniques such as genetic algorithms or variations of weighted A* search, like Explicit Estimation Search (ESS) or Dynamic Weighted A* (DWA*), might offer improvements. [HR14] These approaches, however, would likely require extensive parameter tuning and methodological adjustments to optimize performance.

Lastly, the application of these advanced algorithms to other combinatorially complex problems, such as scheduling and resource allocation, could provide broader benefits and insights. Additionally, testing the scalability of these algorithms in more constrained computational environments would be crucial for evaluating their practical deployment in real-world scenarios. This could help ascertain the versatility and adaptability of the proposed solutions across different domains and challenges.

Appendix A

Heuristic Search Algorithms

Algorithm 13: GBFS

```
Data: $x_0$: Initially Scrambled Cube
Result: Solution or Failure.
1  $OpenList \leftarrow \{x_0\}$;
2  $CloseList \leftarrow \{\ \}$;
3  while $OpenList \neq \emptyset$ do
4      $x \leftarrow$ The node with the lowest $h(n)$ from $OpenList$;
5      $OpenList \leftarrow OpenList \setminus \{x\}$;
6      if $x$ is solved then
7          return $x$'s move sequence
8      $successors \leftarrow$ Expand $x$;
9      for $successor \in successors$ do
10         if $\{successor\} \cap (OpenList \cup CloseList) = \emptyset$ then
11             $OpenList \leftarrow OpenList \cup \{successor\}$;
12     $CloseList \leftarrow CloseList \cup \{x\}$;
13 return Search Failed
```

Algorithm 14: IDA* Search

```
Data: $x_0$: Initially Scrambled Cube
Result: Solution or Failure.
threshold ← $h(x_0)$;
while true do
    result ← Search($x_0$, 0, threshold);
    if result is a solution then
        return result
    else if result is $\infty$ then
        return Search Failed
    else
        threshold ← result;
Function Search($x, g, threshold$)
    $f \leftarrow g + h(x)$;
    if $f >$ threshold then
        return $f$
    if $x$ is solved then
        return $x$'s move sequence
    min ← $\infty$;
    successors ← Expand $x$;
    foreach successor ∈ successors do
        temp ← Search(successor, $g + 1$, threshold);
        if temp is a solution then
            return temp
        if temp < min then
            min ← temp;
    return min;
```

Appendix B

Move Sequence For Permutations and Orientations

Table B.1: Move Sequence For Permuting Corner Cubies To Slot 3

Slots	Moves to Slot 3
Slot 4	`B`
Slot 5	`D D B'`
Slot 6	`D' B'`
Slot 7	`B'`
Slot 8	`D B'`

Table B.2: Move Sequence For Permuting Edge Cubies To Slot 3

Slots	Moves to Slot 3
Slot 4	`B L`
Slot 5	`D D L L`
Slot 6	`D' L L`
Slot 7	`L L`
Slot 8	`D L L`
Slot 9	`R' D D L L R`
Slot 10	`L'`
Slot 11	`L`
Slot 12	`R D D L L R'`

Table B.3: Move Sequence For Generating B for $\ker(f)$

v_i : Vector in Basis	m_{v_i} : Move Sequence to generate v_i
v_1	`F R U R’ R’ D’ R U’ R’ D F’ R U R U’ R’`
v_2	`U R U U R’ L’ U’ L U’ L’ U U L R’ U R`
v_3	`U U R’ U’ R U’ R’ U U R U U R U R’ U R U U R’`
v_4	`F’ U R’ U’ R U’ R’ U U R U U R U R’ U R U U R’ U F`
v_5	`L’ F R U R’ R’ D’ R U’ R’ D F’ R U R U’ R’ L`
v_6	`L U R U U R’ L’ U’ L U’ L’ U U L R U R’ L’`
v_7	`B U U R’ U’ R U’ R’ U U R U U R U R’ U R U U R’ B’`

Table B.4: Move Sequence For Generating B for $\ker(h)$

v_i : Vector in Basis	m_{v_i} : Move Sequence to generate v_i
v_1	`F U U F F D’ U’ L’ U L D F F U’ F’ U’`
v_2	`U R F R U R’ F’ R’ F U’ F F U U F U’ F’ U’ F U’`
v_3	`U’` (m_{v_1}) `U`
v_4	`U R R` (m_{v_1}) `R R U’`
v_5	`F F` (m_{v_1}) `F F`
v_6	`L L` (m_{v_2}) `L L`
v_7	`B B` (m_{v_3}) `B B`
v_8	`U R` (m_{v_3}) `R’ U’`
v_9	`L’` (m_{v_2}) `L`
v_{10}	`L` (m_{v_2}) `L’`
v_{11}	`U’ R’` (m_{v_3}) `R U`

Algorithm 15: Find Scalar Vector of v Based On Given Basis

```
Data: $v$: vector, $b$ : basis
Result: Scalar Vector of Size $|b|$.
1 $Coef \leftarrow$ An array to store the coefficients of the basis vectors;
2 $A \leftarrow$ The augmented matrix such that the columns are the basis vectors;
3 $A \leftarrow [A \mid v]$;
4 Apply Gaussian Elimination to solve the linear system based on $b$;
5 $Coef \leftarrow$ Extract the coefficients from the final form of the matrix;
6 return $Coef$;
```

Appendix C

Experimental Data

C.1 MBWA* and MBS

Table C.1: Average Solution Length By MBWA* and MBS

w B	**1.5**	**1.6**	**1.8**	**2.0**	**2.2**	**2.4**	**2.6**	**2.8**	**3.0**	**5.0**	**8.0**	**MBS**
500	21.51	21.61	22.07	22.53	22.95	23.19	23.47	23.63	23.65	23.65	23.69	23.64 (0.97)
700	21.51	21.61	22.09	22.53	22.93	23.11	23.21	23.41	23.35	23.37	23.37	23.34 (0.98)
1000	21.51	21.61	22.09	22.49	22.81	22.97	23.09	23.15	23.13	23.17	23.15	23.15
1500	21.51	21.61	22.09	22.47	22.71	22.83	22.89	22.93	22.93	22.93	22.93	22.93
2000	21.51	21.61	22.09	22.51	22.65	22.75	22.77	22.85	22.83	22.79	22.81	22.81
2500	21.51	21.59	22.09	22.43	22.57	22.61	22.63	22.63	22.61	22.63	22.63	22.63
3000	21.51	21.57	22.21	22.41	22.51	22.57	22.57	22.59	22.59	22.59	22.59	22.59

Table C.2: Average Solving Time by MBWA* and MBS (s)

w B	**1.5**	**1.6**	**1.8**	**2.0**	**2.2**	**2.4**	**2.6**	**2.8**	**3.0**	**5.0**	**8.0**	**MBS**
500	111.64	70.40	28.95	13.94	8.43	6.20	5.36	4.65	4.34	13.60	286.88	3.95 (0.97)
700	112.03	70.90	29.89	14.83	9.70	7.33	6.56	6.00	5.84	14.81	185.53	5.45 (0.98)
1000	113.47	72.30	31.56	16.02	11.42	9.34	8.49	8.12	8.04	7.89	7.86	7.72
1500	117.03	75.14	34.39	19.02	14.56	12.91	12.11	11.77	11.72	11.69	11.66	11.30
2000	117.85	76.91	36.28	21.03	17.10	15.95	15.59	15.34	15.32	15.26	15.21	14.91
2500	117.52	77.38	37.83	23.50	20.01	18.98	18.62	18.50	18.49	18.42	18.41	18.41
3000	116.50	77.30	38.63	25.89	22.67	21.83	21.54	21.54	21.43	21.39	21.37	22.07

Table C.3: Node Expansion by Batch Size and Scale Factor for MWA* and MBS (in 10^4 units)

B \ w	**1.5**	**1.6**	**1.8**	**2.0**	**2.2**	**2.4**	**2.6**	**2.8**	**3.0**	**5.0**	**8.0**	**MBS**
500	269.65	171.93	72.17	35.05	21.39	15.79	13.67	11.93	11.14	30.65	484.89	10.26 (0.97)
700	271.48	173.81	74.48	37.27	24.54	18.60	16.67	15.30	14.89	33.45	329.13	14.09 (0.98)
1000	274.30	176.73	78.08	40.16	28.74	23.55	21.48	20.54	20.27	19.89	19.86	19.86
1500	278.84	181.62	84.11	46.90	36.09	31.98	30.05	29.23	29.11	28.98	28.98	28.98
2000	283.53	186.76	89.71	52.24	42.65	39.76	38.87	38.28	38.20	38.02	38.01	38.02
2500	288.04	192.05	95.34	59.64	50.80	48.21	47.28	46.98	46.93	46.78	46.77	46.78
3000	293.11	197.42	99.99	67.39	59.00	56.83	56.15	56.03	55.92	55.77	55.76	55.77

C.2 MAWA*

Table C.4: Solution Quality of MAWA* (Compared with MBWA*)

B \ w	**1.6**	**1.8**	**2.0**	**2.2**	**2.6**	**3.0**
500	21.14 (21.61)	21.61 (22.07)	22.02 (22.53)	22.09 (22.95)	22.17 (23.47)	22.31 (23.65)
700	21.23 (21.51)	21.63 (22.09)	22.0 (22.53)	22.07 (22.93)	22.11 (23.21)	22.21 (23.35)
1000	21.16 (21.61)	21.62 (22.09)	22.03 (22.49)	22.07 (22.81)	22.13 (23.09)	22.19 (23.13)
1500	21.16 (21.61)	21.64 (22.09)	22.03 (22.47)	22.07 (22.71)	22.15 (22.89)	22.21 (22.93)
2000	21.16 (21.61)	21.64 (22.09)	22.04 (22.51)	22.07 (22.65)	22.17 (22.77)	22.25 (22.83)
2500	21.11 (21.59)	21.64 (22.09)	22.08 (22.43)	22.07 (22.57)	22.19 (22.63)	22.23 (22.61)
3000	21.01 (21.57)	21.67 (22.21)	22.12 (22.41)	22.13 (22.51)	22.19 (22.57)	22.27 (22.59)

Table C.5: Success Rate of MAWA*

B \ w	**1.6**	**1.8**	**2.0**	**2.2**	**2.6**	**3.0**
500	56	83	98	100	100	100
700	60	84	98	100	100	100
1000	55	81	99	100	100	100
1500	55	81	99	100	100	100
2000	55	81	98	100	100	100
2500	52	81	98	100	100	100
3000	51	81	98	100	100	100

Table C.6: Nodes Expanded of MAWA* ($\times 10^6$)

B \ w	**1.6**	**1.8**	**2.0**	**2.2**	**2.6**	**3.0**
100	1.28	1.31	1.32	1.32	1.30	1.29
200	1.29	1.32	1.32	1.32	1.30	1.29
500	1.31	1.34	1.34	1.34	1.32	1.31
700	1.38	1.41	1.41	1.40	1.38	1.37
1000	1.35	1.38	1.38	1.38	1.36	1.34
1500	1.38	1.41	1.41	1.41	1.39	1.37
2000	1.42	1.45	1.44	1.44	1.43	1.41
2500	1.41	1.44	1.44	1.43	1.42	1.40
3000	1.40	1.43	1.43	1.42	1.41	1.40

Works Cited

[AMSB19] Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8):356–363, Aug 2019.

[Ban82] Christoph Bandelow. *The Mathematical Model*. Birkhäuser Boston, Boston, MA, 1982.

[Bil22] Yakup Bilen. Solving rubik's cube with deep reinforcement learning and a*. https://github.com/yakupbilen/drl-rubiks-cube, 2022. This project aims to solve the Rubik's Cube using deep reinforcement learning and the A* algorithm. It includes a GUI application developed with PyQt5, and the source code is structured to train neural networks, analyze performance, and solve the puzzle with the developed algorithms. Licensed under the MIT License.

[Che] Janet Chen.

[DER18] Erik D. Demaine, Sarah Eisenstat, and Mikhail Rudoy. Solving the rubik's cube optimally is np-complete. 2018.

[DF04] David S. Dummit and Richard M. Foote. *Abstract algebra*. Wiley, New York, 3rd ed edition, 2004.

[DP85] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of a*. *J. ACM*, 32(3):505–536, jul 1985.

[ED09] Rüdiger Ebendt and Rolf Drechsler. Weighted a* search – unifying view and application. *Artificial Intelligence*, 173(14):1310–1342, 2009.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[HR14] Matthew Hatem and Wheeler Ruml. Simpler bounded suboptimal search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 28(1), Jun. 2014.

[HZ07] E. A. Hansen and R. Zhou. Anytime heuristic search. *Journal of Artificial Intelligence Research*, 28:267–297, March 2007.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[KC07] Daniel Kunkle and Gene Cooperman. Twenty-six moves suffice for rubik's cube. In *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*, ISSAC '07, page 235–242, New York, NY, USA, 2007. Association for Computing Machinery.

[Kor97] Richard E. Korf. Finding optimal solutions to rubik's cube using pattern databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, page 700–705. AAAI Press, 1997.

[Lee18] G.T. Lee. *Abstract Algebra: An Introductory Course*. Springer Undergraduate Mathematics Series. Springer International Publishing, 2018.

[MASB18] Stephen McAleer, Forest Agostinelli, Alexander Shmakov, and Pierre Baldi. Solving the rubik's cube without human knowledge, 2018.

[Riv23] Riverbank Computing. PyQt5. https://riverbankcomputing.com/software/pyqt/, 2023.

[RKDD14] Tomas Rokicki, Herbert Kociemba, Morley Davidson, and John Dethridge. The diameter of the rubik's cube group is twenty. *SIAM Review*, 56(4):645–670, 2014.

[RN20] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 4 edition, 2020.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning series. The MIT Press, Cambridge, Massachusetts, second edition, 2018.

[Shr22] Anshumali Shrivastava. Deep learning. Lecture Notes in COMP 642 – Machine Learning, 2022.

[Tak23] Kyo Takano. Self-supervision is all you need for solving rubik's cube, 2023.

[Thi81] M.B. Thistlethwaite. The 45-52 move strategy. *London CL VIII*, 182, 1981.

[Wor24] World Cube Association. World cube association results database export. Online, April 2024.

[YLX$^+$10] Junfeng Yao, Chao Lin, Xiaobiao Xie, Andy JuAn Wang, and Chih-Cheng Hung. Path planning for virtual human motion using improved a* star algorithm. In *2010 Seventh International Conference on Information Technology: New Generations*, pages 1154–1158, 2010.