State-Value Estimation of a 3x3 Rubik's Cube

Landon Harris, Coby White
University of Tennessee Knoxville
{lharri73, cwhit152}@vols.utk.edu

Abstract— **For this project, we aim to solve the Rubik's Cube puzzle with reinforcement learning. This comes with many challenges, namely that there are 4.325×10^{19} different possible states of any one cube, so using standard value iteration would be infeasible. Instead, we explored function approximation methods including Deep Q-Learning and state-value estimation. We used a mix of human-targeted algorithms such as Fridrich's method to train our state-value estimation network to expedite the training process. We also used algorithms designed for machines such as Two-Phase to improve on human targeted algorithms and improve the cost-to-go function. With a trained network, we compare a greedy policy with a policy founded on the popular A* shortest-path method, using our state-value network as the heuristic function.**

I. Introduction

In 1974, Erno Rubik wanted to find a way to model three-dimensional movement to his students. After months of work, he a created 3x3 cube made out of wood and paper, held together by rubber bands, glue, and paper clips. He called this 3x3 cube *The Magic Cube* [1]. The Magic Cube was eventually renamed to the Rubik's Cube, and it quickly became the most popular puzzle toy in the world, with 350 million Rubik's Cubes sold by 2018 [1]. Since its inception, people have been challenged to solve the Rubik's Cube's large state space, with 4.235×10^{19} possible ways to permute the cube[2], as quickly and efficiently as possible. Given the Rubik's Cube's large state space, the algorithms used to generate solutions are generally heuristic and can be broken up into two categories: those meant to be solved by humans (known as speedcubers), and those meant to be solved by machines (the focus of this work).

Speedcubers are a niche group of individuals who challenge each other to solve a random scramble of a Rubik's Cube as fast as possible, more specifically, faster than anyone else. Within the speedcubing community there are three main algorithms used to solve the Rubik's Cube: Fridrich's Method, Roux, and ZZ [3]. Each method has their own pros and cons, but Fridrich's Method is the most popular method used within the speedcubing community [3]. These methods, however, are typically targeted at being simple to remember and easy for fingers to swiftly move the Rubik's Cube, often generating long sequences to solve a scramble. The most skillful of speedcubers can solve the Rubik's Cube in roughly fifty to sixty moves [4].

Using a machine to solve a Rubik's Cube uses a different set of algorithms, including the use of pattern databases[5], Deep Reinforcement Learning (DRL) [6], and even evolutionary algorithms[7]. Further, each implementation has its own representation of the Rubik's Cube and is limited by how it can manipulate the cube. For example, OpenAI used a human-like robotic hand[6], and a common solver/scrambler tool can only manipulate 5 of the 6 sides[8].

For this problem, classic tabular Reinforcement Learning (RL) methods cannot be used because it would require roughly 1.8 Zetabytes[1] to store the table. Instead, we use DRL to solve a Rubik's Cube with function approximation, eliminating the need to visit each state and evaluate the action-value function, but also solves the intractable problem of storing this large table.

II. Previous Work

Algorithms used to solve Rubik's Cubes by a machine generally require hand-engineered methods and group theory to solve[9]. One of the more popular solving algorithms to solve the Rubik's Cube is Kociemba's Two-Stage Solver [10]. The Kociemba Two-Stage Solver takes advantage of the Rubik's Cube's group properties to create smaller sub-groups which then makes the task of solving it trivial [10]. Korf is a more heuristic approach to solve the Rubik's Cube, using a different type of A* heuristic search as well as a pattern database heuristic to find the shortest possible path to the solution [5]. Now, scientists and researchers are wanting to use deep neural networks using supervised learning alongside feature engineering to create these heuristics. However, those algorithms take a long time to run and generally do not solve the Cube within a reasonable amount of time [10].

Researchers from University of California, Irvine have figured out a way to solve Rubik's Cubes without human knowledge using DRL [10]. The researchers developed an algorithm called Autodidactic Iteration (ADI), inspired by policy iteration, and created to overcome the *sparse reward problem* in a model-based environment with a large state space. ADI was used to train a joint value-policy network in a supervised fashion. Once the training is complete, the network is then combined with a Monte Carlo Tree Search to efficiently solve the Rubik's Cubes. This solver was dubbed by the researchers as DeepCube [10].

The DeepCube group continued their work, and have now been able to advance it to what has been called DeepCubeA [11]. DeepCubeA combines both deep learning with classical reinforcement learning (approximate value iteration) and path finding methods. DeepCubeA works by training a deep neural network to approximate a function that then outputs the

[1]Assuming 32 bit floats, storing the action-value table for all states would require $(4.235 \times 10^{19} \times 12 \times 32)$ bits

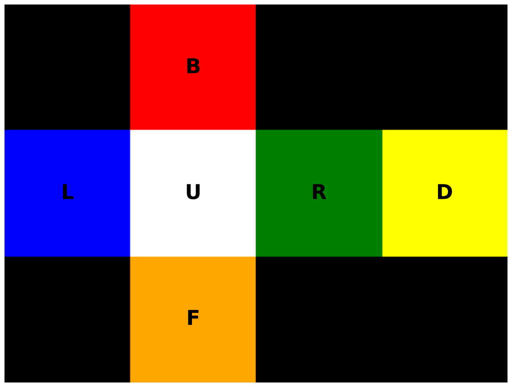

Fig. 1. Net representation of a Rubik's Cube used in our environment.

cost to reach the goal state, used as the state-value function. The system trains on states obtained by starting from the goal state and randomly taking moves in reverse[11]. This method uses an optimal solver to compute the ground-truth cost-to-go for each state, which can take hundreds of seconds to compute for some states[12]. This method also implements a heuristic which hard-codes a state-value of 0 for states that are 1 step away from the goal state (including the goal state).

III. Domain Problem

A common Rubik's Cube has 6 faces, shown in fig. 1. A Rubik's Cube's center pieces do not move without reorienting the cube itself (something that is not possible with the actions available in our environment), so we require that the cube maintain the orientation shown in fig. 1; that is white $\rightarrow$ up, blue $\rightarrow$ left, yellow $\rightarrow$ down, etc.

Actions on a Rubik's Cube are represented by the face the player wishes to rotate. So a clockwise rotation of the right face would be represented as **R**, and a counterclockwise rotation would be **R'**. To make notation easier, the speedcubing community has added a third notation for double rotations, so rotating the right face twice (clockwise or counterclockwise) would be represented as **R2**. For this project, we do not allow double rotations in a single action, so this would require a string of two actions. It should be noted that there are two possible strings of actions that can result in the same state as a single **R2** action, those are **R', R'** or **R, R**.

Although the state space for a Rubik's Cube is large, as discussed previously, it can still be represented as a Markov Decision Process (MDP). Our state is represented by a three-dimensional matrix with shape $(3 \times 3 \times 6)$ where the first dimension indexes the rows, second indexes the columns, and the third indexes the *side* or *face* of the Rubik's Cube. This maintains the Markov property because future states are not dependent on the *path* that we took to reach the current state, or in other words, there are many different ways to reach the same state, each independent of how you got there.

Our reward structure is simply the number of actions that state is away from being solved, as determined by Kociemba's TwoPhase algorithm[13].

IV. Reinforcement Learning Methods

For this project, we planned (and proposed) to used Deep Q-Learning (DQN). DQN is an extension to classical Q-learning that uses a neural network to approximate the action-value function, known as the Q-function. The goal of Q-learning is to learn a policy that tells the agent what action to take in a given state in order to maximize the expected reward. In DQN, the Q-function is approximated using a deep neural network, allowing the algorithm to work with high-dimension state spaces and learn more complex policies[14]. We found this idea attractive for this problem given the Rubik's Cube's large state space, but found during training that regular DQN was not converging at all. We hypothesize that this is because we were requiring the network to learn how many steps away from the solution, not only the current state was, but each resultant state from every possible action (it is not always the case that a suboptimal action raises the cost-to-go by one and the optimal action lowers the cost-to-go by one).

Instead we chose to follow the idea of [11] and train a network to predict the state-value of each state. This reduces the complexity of the training process, but increases the complexity during inference. We explored two policies generated based off of this state-value function: a greedy policy that takes the action which leads to the state with the lowest state value, which can be thought of as an estimate of the number of moves required to solve; and a policy based on an A* weighted search, where the heuristic for each node is the state-value prediction from the network. While the greedy policy requires fewer network inferences, it can get stuck oscillating among a subset of states, leading to an infinite loop. The A* policy eliminates this cycling issue, viewing the state-space as a graph and finding the shortest path to the solution.

V. Code Design

Fridrich's method, as mentioned earlier, is a common way speedcubers solve Rubik's Cubes. We used CubeLuke's implementation of Fridrich's Method [15] found on GitHub. Although we used this method and implementation to solve the Rubik's Cube; we were truly only concerned with how many moves would be required to solve the Rubik's Cube not which moves were taken (after we ensured that it was accurate, of course). Since this was a method targeted for humans rather than machines, the amount of steps required to solve the Rubik's Cube is typically much larger (generally in the range of 50-60 moves) than the optimal solution. This comes at the benefit of being able to reach the solution in a few milliseconds.

We also used Kociemba's Two-Phase algorithm[13] to formulate the cost-to-go function. The Two-Phase solver would always solve the Rubik's Cube in substantially less moves than Fridrich's Method, generally in around 20 moves. However Two-Phase has a substantially longer run time than

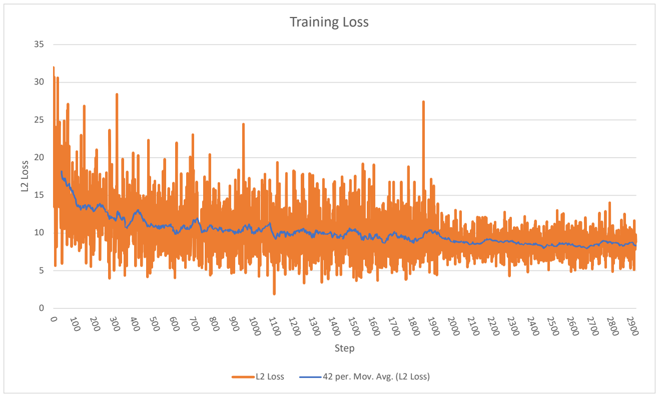

Fig. 2. Loss during training of the state-value model.

Fridrich's Method. We implemented a timeout period that kept the Two-Phase solver from taking longer than three seconds to solve, if it did, the cost-to-go function uses the longer (move-wise) solution provided by Fridrich's Method.

For an efficient, modular implementation of A*, We also used a package created by Julien Rialland [16]. This was used to create the A*-based policy.

We created an OpenAI Gym [17] environment to represent the Rubik's Cube with utility functions to translate the state across the two solvers obtained on GitHub and a visualization function that creates nets shown in fig. 1.

Our model is inspired by that of the [11], using a Resnet model[18] with 2 hidden layers, with 5000 nodes and 1000 nodes respectively, followed by 4 residual blocks where each residual block has two hidden layers of size 1000. The output layer consists of a single linear unit, this represents our estimated cost-to-go. We trained our model using a batch size of 32, where each sample in the batch was a cube scrambled randomly with n displacements where n is sampled uniformly on the closed interval $[0, 25]$. This made it highly unlikely that the model ever saw the same scramble twice with higher values of n. We trained the model for 2 days on an Nvidia A6000 until training stalled, shown in fig. 2.

VI. Results Analysis

A. Accuracy

We conducted a variety of experiments to determine the efficacy of our approach. The most insightful was determining the error rate of the model across different scramble lengths. The results of our experiment are shown in fig. 3, and are still puzzling[2] to us. We expect the cost-to-go *error* to increase as the complexity of the cube increase and relationships among the sides become more difficult to find. This expectation holds for scrambles with displacements up to around 13, but then the error starts to decrease as the Rubik's Cube scramble gets more complicated. We initially thought this was because as we increased the number of displacements in a solve, we were taking the Rubik's Cube to a state that was

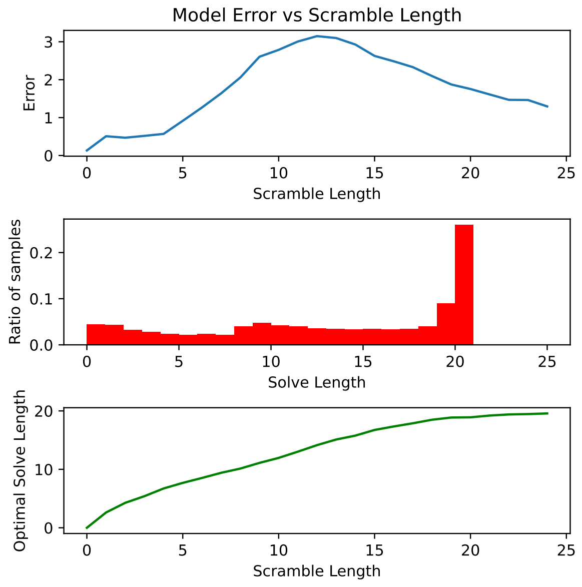

Fig. 3. *Top:* Model evaluation comparing the accuracy of the state-value estimation model to the TwoPhase algorithm's cost-to-go result over Rubik's Cubes scrambled by up to 25 displacements. *Middle:* Distribution of the number of moves required to solve the cube over this experiment (regardless of number of displacements during initial scramble). *Bottom:* Average length of the optimal solve as the scramble length increases. All averaged across 100 scrambles for each length.

not at the most scrambled state. In other words, we thought that by further scrambling the cube (after 13 displacements), we could be reducing the cost-to-go. We found that this was not actually the case by graphing the average cost-to-go for each scramble length, and notice it steadily increases to 20 over the range of our displacement test, shown in the bottom of fig. 3.

Our final approach to explain this decrease in error after 13 displacements was to explore the hypothesis that the mean optimal solution for any scramble high. It has been proven that the max number of moves required to solve any scramble is 20[19], but we suspected that the optimal solve length that occurred most often was also on the higher end, after the peak of the error curve. After plotting the distribution of solve length across the entire experiment, shown in the middle of fig. 3, we found the most common solve length to be 20, accounting for nearly 25% of scrambles. This begins to explain why the network would be more capable of predicting the cost-to-go at states that have higher scrambles (because it was exposed to states with cost-to-go values of 20 more often during training), but doesn't explain the peak at 13.

B. Efficiency

We explored two different policies based on this state-value estimation: a greedy algorithm that simply takes the action that results in a state with the lowest cost-to-go; and a policy that employs the A* algorithm to reduce cycles and achieve the shortest path with a more global view. The greedy

[2]get it...puzzling?

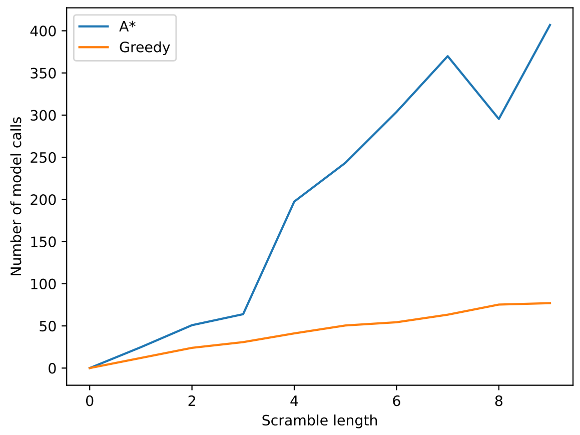

Fig. 4. Number of model inferences using an A* policy and a Greedy policy as scramble length increases, averaged across 50 scrambles for each length.

algorithm suffers from cycling among a subset of states and never converging, but requires fewer model inferences as we only run an inference 12 times (once for each possible action), then take the action, and repeat until we find the goal. The A* algorithm builds a graph, querying the heuristic function (and in turn, the model), in a breadth-first, or really, best-first fashion. The results of this experiment are shown in fig. 4. As the number of displacements in the scramble increases, the number of model inferences using the greedy method increases linearly. A*, on the other hand, increases seemingly exponentially as it has to evaluate the heuristic function for every state it encounters to determine which state is closer to the goal, especially if it has to go back when it encounters a cycle.

VII. Conclusion

We were able to achieve a Rubik's Cube using a form of DRL. While the greedy policy gets stuck in cycles and the A* policy has a tendency to visit a large amount of states, between both methods, we are able to solve 73% of scrambles with the optimal solution. If we had spent more time training and/or tuning our network, we believe that we would have been able to increase this to solve nearly all scramble with the optimal solution. We also explored an unconventional policy method in the form of A* search, which proved to be an effective method at finding the shortest path to the solution. While we still have some unanswered questions, we've enjoyed working on this project and believe it has been both challenging yet entertaining to attempt to employ Deep Reinforcement Learning to solve a puzzle that has been around since the advent of the personal computer.

References

[1] S. Magazine, "A brief history of the rubik's cube." [Online]. Available: https://www.smithsonianmag.com/innovation/brief-history-rubiks-cube-180975911/

[2] M. Gymrek and J. Li, "The mathematics of the rubik's cube," Mar 2009. [Online]. Available: http://web.mit.edu/sp.268/www/rubik.pdf

[3] [Online]. Available: https://www.youtube.com/watch?v=QKK8J3JKWi4

[4] Tamar, "How many steps does it take to solve a rubik's cube? – gocube." [Online]. Available: https://getgocube.com/play/steps-to-solve-rubiks-cube/

[5] R. E. Korf, "Finding Optimal Solutions to Rubik's Cube Using Pattern Databases," p. 6.

[6] Oct 2019. [Online]. Available: https://openai.com/blog/solving-rubiks-cube/

[7] J. J. Grefenstette, D. E. Moriarty, and A. C. Schultz, "Evolutionary algorithms for reinforcement learning," *Journal of Artificial Intelligence Research*, vol. 11, p. 241–276, Sep 1999, arXiv:1106.0221 [cs]. [Online]. Available: http://arxiv.org/abs/1106.0221

[8] [Online]. Available: https://www.gancube.com/GAN-ROBOT

[9] [Online]. Available: http://kociemba.org/math/imptwophase.htm

[10] S. McAleer, F. Agostinelli, A. Shmakov, and P. Baldi, "Solving the rubik's cube without human knowledge," *CoRR*, vol. abs/1805.07470, 2018. [Online]. Available: http://arxiv.org/abs/1805.07470

[11] F. Agostinelli, S. McAleer, A. Shmakov, and P. Baldi, "Solving the rubik's cube with deep reinforcement learning and search," *Nature Machine Intelligence*, vol. 1, no. 8, p. 356–363, Aug 2019. [Online]. Available: https://www.nature.com/articles/s42256-019-0070-z

[12] H. Kociemba, "Rubikscube-optimalsover." [Online]. Available: https://github.com/hkociemba/RubiksCube-OptimalSolver

[13] ——, "Rubikscube-twophasesolver." [Online]. Available: https://github.com/hkociemba/RubiksCube-TwophaseSolver

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[15] Lucas and T. Brannan, "Rubiks-cube-solver," Jan 2020. [Online]. Available: https://github.com/CubeLuke/Rubiks-Cube-Solver

[16] J. Rialland, "python-astar." [Online]. Available: https://github.com/jrialland/python-astar

[17] [Online]. Available: https://www.gymlibrary.dev/

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[19] T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge, "The diameter of the rubik's cube group is twenty," *SIAM Review*, vol. 56, no. 4, pp. 645–670, 2014. [Online]. Available: https://doi.org/10.1137/140973499