Guiding Rubik's Cube Reinforcement Learning Algorithms for Interpretability

Cheng Chang[*1] Terry Feng[*2] Jerry Liu[*1]

Abstract

The Rubik's cube is a challenging policy exploration environment for reinforcement learning (RL) methods due to the sheer size of its state space and sparse reward. We build upon DeepCubeA, an RL approach with a deep value network and A* search, to develop a model that balances solve optimality and desired interpretability. Goals for interpretability include substructure stability or enforcing a desired multi-stage approach. We propose a simple but generalizable method for reward shaping in the context of the Rubik's cube using correctness indicator functions on individual pieces, and demonstrate that our reward shaping framework is an effective means of achieving this structure. We provide evidence for a tradeoff between minimizing solve lengths and longer but more interpretable solutions, and we demonstrate that our framework has the capacity to learn algorithms that interpolate between these two extremes. Empirically, we train models based on the layer-by-layer and two-stage corners-then-edges approaches, two human methods to solve the Rubik's cube, and we show that our models do indeed learn to preserve human-desired substructures within their solution trajectories.

1. Introduction

The Rubik's cube is a combinatorial optimization environment with a huge number of possible states (43 quintillion) but only a single reward state. Since the reward signal is sparse and episodes are unlikely to end through random chance alone, this setting is challenging for typical RL algorithms.

Much work has been done to create cube-solving algorithms that find near-optimal solutions to any cube state within a reasonable amount of time. Most of the recent methods that achieve this goal to a reasonable extent use deep reinforcement learning (DRL). Generally, these methods focus on reducing the number of moves used to solve the Rubik's cube from any given position. However, there is a tradeoff between move efficiency and algorithm interpretability. In general, it's not possible for a human to learn how to solve the Rubik's cube by trying to emulate a DRL algorithm since there are no human-perceptible measures of progress throughout the solve.

Most human approaches involve multiple independent stages and subroutines (typically only influencing a small fraction of the pieces) that are used throughout the algorithm. One paradigmatic example is the beginner's method[1], also called the layer-by-layer method (LBL). Using this method, the solving process is broken into the 3 phases, solving the bottom layer, inserting the middle layer edges, and orienting and permuting the top layer. These three phases allow for a simple, human-understandable reward signal (visual incremental progress) and represent a natural way a human may try to solve the Rubik's cube. Another human cube-solving method with easily measurable substructures is the corners-then-edges method[2] (CTE), whose name suggests the two phases during the solving process. In addition, to be understandable, this more structured way of approaching the problem can be transferred to build intuition to solving the Rubik's cube in more difficult settings (the 4x4 and 5x5 cubes, for example).

In this project we explore whether, instead of optimizing over solution length, reward shaping inspired by a human-like approach can help DVNs develop human-understandable algorithms for solving the Rubik's cube. We focus on understanding whether guiding the cube-solving algorithm with human-like reward signals would achieve this goal. In existing DRL methods, trajectories are generated starting from the solved state with random scrambles, which allows a value network to learn an approximation for the minimum number of moves required to solve the cube from a given scrambled position. Another natural metric

[*]Equal contribution [1]Institute for Computational and Mathematical Engineering, Stanford University, Stanford, CA, USA [2]Center for Computer Research in Music and Acoustics, Stanford University, Stanford, CA, USA. Correspondence to: Skanda Vaidyanath <svaidyan@stanford.edu>.

[1]https://ruwix.com/the-rubiks-cube/how-to-solve-the-rubiks-cube-beginners-method/

[2]https://www.speedsolving.com/wiki/index.php/Corners_First

to consider is the number of pieces currently in the correct position. This is not only a more reasonable metric for a human to use since it is easily perceptible, but it also has finer-grain reward signal. We are interested in whether training a DRL network using these rewards may be quicker and may converge to a more human-like algorithm.

In particular, we develop a framework for reward shaping for the Rubik's cube that is generalizable enough to smoothly interpolate between algorithms that optimize for minimum number of moves and algorithms with hard constraints that follow human-desired guidelines, such as solving the Rubik's cube in multiple independent stages. Training a couple of models along this spectrum, we demonstrate that our resulting algorithms do indeed preserve the substructures we want within their solution trajectories, and we provide evidence that enforcing human-interpretability constraints acts as a form of regularization during the actual solve that reduces variance in solution trajectory lengths and in number of nodes explored.

2. Related Work

Several works spanning decades have developed computer algorithms for solving the Rubik's cube. The earliest works exploit human knowledge about the system via group-theoretic guarantees to reduce the problem space and use look-up tables and search algorithms, such as those used in Kociemba (1992) and Korf (1997). These methods are guaranteed to find optimal or near-optimal solutions in terms of the number of moves taken (called "God's Number", which is 20 for the 3x3 Rubik's Cube[3]), but they involve visiting a huge amount of nodes during searching, usually on the order of 10^{10} to 10^{11}, or 10 to 100 billion (Kociemba, 1992; Korf, 1997). While it is a great improvement from simply searching through all states, the time complexity required by these algorithms is high for non-theoretical use.

More recently, Deep Reinforcement Learning (DRL) algorithms have been developed that solve the Rubik's cube more efficiently (Brunetto & Trunda, 2017; McAleer et al., 2018; Agostinelli et al., 2019). The earliest attempt to train a neural network that provides useful heuristics to search algorithms is from Brunetto & Trunda (2017). This approach introduced backward state generation, which provided training data to a network by scrambling a solved Rubik's cube one step at a time and recording the value of these states based on a human-defined metric. Later, McAleer et al. (2018) drew from this training procedure to develop what they call Autodidactic Iteration (ADI), which is a value iteration algorithm that trains a deep value network (DVN) by repeatedly updating it with a depth-1 greedy tree search policy using the current DVN. Building on this, they produce the DeepCubeA algorithm, which consists of a DVN trained by a simpler value iteration and a batched weighted A* Search (BWAS), which greatly reduced solve time complexity and produced solutions with near-optimal moves (Agostinelli et al., 2019).

[3]http://www.cube20.org/

2.1. DeepCubeA

We base our implementation on the code provided by (Agostinelli et al., 2019), DeepCubeA[4]. We give a brief description of their training paradigm here because it is highly relevant to our intended way of approaching the problem.

To represent each state of the 3x3 cube, DeepCubeA uses an integer array of size 54, 9 squares times 6 faces, with each element indicating the color of a specific position on a specific face of the cube. Note that this means each physically possible state of the cube is represented as a unique array. All 12 of the possible cube turns are simplified as hard-coded permutations of the integer array. DeepCubeA's neural network, consisting mainly of alternating linear and batch normalization layers, is trained to take in any scrambled state and return the *cost-to-go*, the approximate minimum number of moves required to solve the cube from this position. Since there is no guarantee that this network's output is exactly correct, DeepCubeA treats this as a "heuristic function", to be used in conjunction with provably correct shortest paths algorithms, such as A* search.

To tackle the sparse reward problem, DeepCubeA first generates data by starting from the solved state and applying random moves until reaching a suitably scrambled state. From here, a limited A* search is performed on the trajectory's nodes using the current heuristic network outputs as the heuristic function. The loss that the heuristic network uses to update its weights is the sum of the Bellman error. Specifically, for a given state s with adjacent states $N(s)$, the loss at this state is

$$L(s) = \begin{cases} h(s)^2 & s \text{ solved} \\ (h(s) - (\min_{s' \in N(s)} h(s') + 1))^2 & \text{else} \end{cases} \tag{1}$$

where $h(\cdot)$ is the heuristic network and $N(s)$ are the neighboring states of s. Note that this procedure is equivalent to (deep) value iteration, except that our goal is to minimize the value rather than maximize it, and Bellman updates are only performed on a small subset of the possible states per weight update, so the value iteration is approximate.

In order to solve the Rubik's cube using the fully-trained heuristic network, a full A* search is performed from the given scrambled state using the heuristic network's outputs as estimates for the true distances to the solved state.

[4]https://github.com/forestagostinelli/DeepCubeA

3. Approach

3.1. Reward Shaping Framework

We define a general, simple framework for reward shaping based on state-dependent penalties and Markovian transition costs.

Recall that DeepCubeA's training objective seeks to train a heuristic function $h(s)$ approximating the minimum number of moves required to get from state s to the solved state by penalizing a squared loss on the Bellman error (1). This can be rewritten in the form

$$\min_{\theta}(h_\theta(s) - (\min(c(s), \min_{s' \in N(s)} h(s') + 1 + \gamma T(s, s'))))^2$$

where $c(\cdot) : S \to \mathbb{R}_+ \cup \{\infty\}$ is a penalty function and $T(\cdot, \cdot) : S \times S \to \mathbb{R}_+ \cup \{\infty\}$ is a transition cost function. Note that in our definition, the hyperparameter γ controls the extent to which we want to optimize for fewest number of moves vs. our desired constraints on the algorithm.

In DeepCubeA,

$$c(s) = \begin{cases} 0 & s \text{ solved} \\ \infty & \text{else} \end{cases}$$

and $\gamma = 0$.

To bias the behavior of the model away from optimizing for the fewest moves and toward a more human-desirable strategy, we define our intermediate reward by checking which of the edges and corner pieces are in the correct place and/or are oriented correctly. In particular, define the indicator function on piece $p \in P$ as

$$\mathbb{1}_p(s) = \begin{cases} 1 & \text{piece p in correct location \& orientation} \\ \delta_p & \text{piece p in correct location only} \\ 0 & \text{piece p not in correct location} \end{cases}$$

Here we have hyperparameters $0 \leq \delta_p \leq 1$ that encode how much we care about orienting vs. fully solving piece p.

We consider $T(\cdot, \cdot)$ restricted to be within the function space of linear combinations of these indicator functions:

$$T(s, s') = \sum_{p \in P} \left(\lambda'_p \mathbb{1}_p(s') - \lambda_p \mathbb{1}_p(s)\right)$$

In the following examples, we show that this framework for reward shaping is flexible and powerful enough to induce human-desirable substructures within the learned algorithms.

3.2. Layer-by-Layer

The layer-by-layer (LBL) algorithm is one of the most beginner-friendly human methods for solving the Rubik's cube because it has an easy progress metric and requires relatively few algorithms. The approach is to solve the cube in three stages: first solve all the pieces in the top layer, then solve the pieces in the middle layer while preserving the top layer, then finally solve the bottom layer while preserving the top and middle layers.

Trying to directly enforce the three-stage approach as a hard constraint is difficult because it is difficult to characterize the set of subroutines (sequences of moves) that can preserve the results of previous stages while making progress within the current stage. As a result, we instead define a soft-constrained version of LBL within our reward-shaping framework. We set

$$c(s) = \begin{cases} 0 & s \text{ solved} \\ \infty & \text{else} \end{cases}$$

as in DeepCubeA, and we define the transition costs as

$$T(s, s') = \sum_{i \in \{1,2,3\}} \frac{\lambda_i}{|L_i|} \sum_{p \in L_i} (\mathbb{1}_p(s) - \mathbb{1}_p(s'))$$

where L_1, L_2, L_3 represent the top, middle, and bottom faces respectively. This penalizes moves that disrupt the existing progress in each layer: for example, a move that takes the cube from a completely solved L_i to a completely unsolved L_i (while leaving the other layers untouched) incurs an extra transition cost of λ_i. When enforcing

$$\lambda_1 > \lambda_2 > \lambda_3 > 0$$

we expect the algorithm should be incentivized to prefer solving the top layer before the middle layer and before the bottom layer whenever expedient, and the degree of preference can be controlled by tweaking the relative magnitudes of the λ values. Note that fixing $\frac{\lambda_1}{\lambda_2} = k_1 \gg 1$, $\frac{\lambda_2}{\lambda_3} = k_2 \gg 1$, and taking $\lambda_1, \lambda_2, \lambda_3 \to \infty$, the result of the optimal algorithm approaches the hard-constrained three-stage version of LBL.

3.2.1. LBL: EXPERIMENTAL SETTINGS

In our experiments, we set $\lambda_1 = 1, \lambda_2, \lambda_3 = 0$, and $\gamma = 0.9$. We finetune our model from the existing DeepCubeA model (equivalent to $\gamma = 0$) for 0.5 million additional iterations. As a result, our results may be different from a true LBL model trained from scratch. We evaluate our model on 10 scrambled positions, which takes us 4 GPU days and 40 CPU days to run. The results of our LBL experiments can be found in Section 4.1.

3.3. Two-Stage Corners-then-Edges

Here we describe the process we use to encode the two-stage corners-then-edges (CTE) algorithm using our reward-shaping framework.

3.3.1. Corners-only

Using our reward-shaping framework, we train a model to solve only the corners of the Rubik's cube. We set

$$c(s) = \prod_{p \in C} \frac{1}{\mathbb{1}_p(s)} = \begin{cases} 0 & \text{corners solved} \\ \infty & \text{else} \end{cases}$$

where C represents the set of corner pieces and

$$T(s, s') = -\sum_{i \in \{1,3\}} \frac{\lambda_i}{|C_i|} \sum_{p \in C_i} (\mathbb{1}_p(s'))$$

where C_i represents the set of corner pieces in layer L_i.

Note that since we do not place any constraints on the status of the edge pieces within the state cost function or within the transition cost function, all positioning of edges are equally valid. Thus, this stage of the algorithm is equivalent to solving a 2x2 Rubik's cube with centers, which has a significantly reduced state space (about 88 million states[5]) compared to the full 3x3 cube.

Setting $\lambda_1 > \lambda_3 > 0$ allows us to also enforce a layer-by-layer-like constraint within the corners-only solving process.

3.3.2. Edges given corners-only

To train a second model to solve the full Rubik's cube given a state with all corners solved, we set

$$c(s) = \begin{cases} 0 & s \text{ solved} \\ \infty & \text{else} \end{cases}$$

as in DeepCubeA and

$$T(s, s') = -\sum_{p \in C} \frac{\lambda_C}{|C|} \mathbb{1}_p(s') - \sum_{p \in E} \frac{\lambda_E}{|E|} \mathbb{1}_p(s')$$

where C and E are the sets of corner and edge pieces respectively. Here, the transition cost increases whenever the new state s' has fewer solved corners or edges. Since the input to this second model is the solved state of the corners-only model, the transition cost incurs a penalty of

$$\lambda_C \frac{\text{number of corners displaced}}{|C|}$$

whenever a move is made that rescrambles the corners. Choosing $\lambda_C \gg \lambda_E$, the model is heavily discouraged from rescrambling any of the corners. Indeed, fixing $\frac{\lambda_C}{\lambda_E} = k \gg 1$ and taking $\lambda_C, \lambda_E \to \infty$, the optimal algorithm converges to a hard-constrained two-stage corners-then-edges approach.

[5]https://artofproblemsolving.com/wiki/index.php/Rubiks_cube

3.3.3. CTE: experimental settings

In our experiments, we set $\lambda_1 = 1$ and $\lambda_3 = 0$ in the corners-only model and evaluate the effects of different settings of $\gamma \in \{0, 0.5, 0.9, 1\}$. Due to the significantly smaller state space, we train for 0.3 million iterations per model, 500 states per iteration. We evaluate 100 solution trajectories for each model.

Due to computational limitations, we finetune an edges-only model with $\lambda_C = 0.9, \lambda_E = 0.1$ based on the original DeepCubeA model for 0.5 million iterations, 500 states per iteration. We found that this model performed almost identically to the original DeepCubeA, likely due to the limited training data. Our results can be found in Section 4.

3.4. Subroutines

We were also curious to explore whether adding human-developed subroutines, or well-defined sequences of actions, to the action space might help the training or performance of the models. Since these subroutines typically swap or rotate a couple of pieces only, they help generate more interpretable solutions. Unfortunately, we found that adding subroutines (and all of their equivalent forms on different faces) increased the number of possible trajectories exponentially. For example, adding a single edge-swap algorithm on every face with left and right symmetries meant adding 24 extra moves. When generating data, this resulted in a significantly reduction in training speed that was computationally intractible.

4. Results

Remark: We focus most of our results, including when investigating the effects of our soft LBL approach, on the corners-only stage of the CTE method. This is because computational cost was a major limiting factor for our experiments. For reference, the original DeepCubeA model was trained for 1 million iterations with 5000 states per iteration, taking 4 GPU weeks and 30 CPU weeks. As mentioned in Sections 3.2.1, 3.3.3, due to limited computational resources, we struggled to train any models on the full 3x3 environment, even the original DeepCubeA model. We found that reducing the states per iteration heavily worsened the performance of our models, and that any of our models operating on the full 3x3 Rubik's cube environment needed to be trained for a similar number of iterations as the original DeepCubeA for a fair comparison. As a result, the limited full Rubik's cube results we have are with models finetuned from the original DeepCubeA weights.

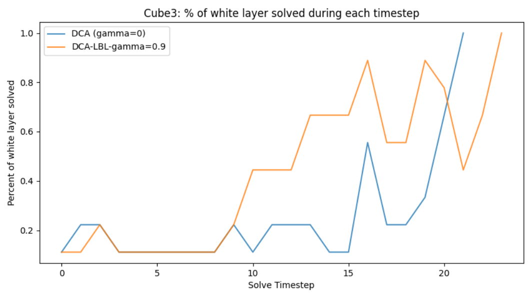

Figure 1. Percent of top layer solved during a 3x3 solve trajectory comparing DCA and DCA-LBL.

4.1. Full 3x3: Layer-by-Layer Results

We successfully solved ten 3x3 Rubik's cube scrambles using our DeepCubeA + layer-by-layer (DCA-LBL) implementation. Comparing the performance of this model against our baseline DeepCubeA (DCA) model, we found that many solve trajectories were similar in both length and interpretability. However, two solve trajectories showed improved interpretability induced by our LBL intermediate rewards. Figure 1 is an example of one of these solve trajectories. Note that as expected, our LBL reward shaping framework does indeed bias our solve trajectory toward solving the top layer earlier, and that this progress is maintained through the remainder of the solve. This can be seen visually as well (Figure 2): DCA's solution trajectory has no clear trends until the cube becomes solved in the final few moves, whereas DCA-LBL's trajectory clearly shows the white (top) face remaining mostly solved throughout.

In this example, DCA-LBL returns a solve trajectory that makes a tradeoff between move efficiency and prioritizing our layer-by-layer substructure and progress. DCA-LBL's solution is two moves longer than DCA's yet incurs less penalty.

4.2. Corners-only: Layer-by-Layer Results

We focus our results on the corners-only stage of our corners-then-edges approach (similar to a 2x2 cube environment). As mentioned in 3.3.3, we evaluate different LBL hyperparameters within this stage for computational efficiency. This streamlines computation for our layer-by-layer approach and establishes our corners-then-edges framework. As our training had yet to fully converge (Figure 3), we present promising early results.

In our LBL approach, our reward-shaping framework encouraged DCA to prioritize solving the top (white) face of the Rubik's cube while solving the entire cube. When $\gamma = 0$ this means no additional reward was given for solving the white face (DCA) and when $\gamma = 1$, this means that solve

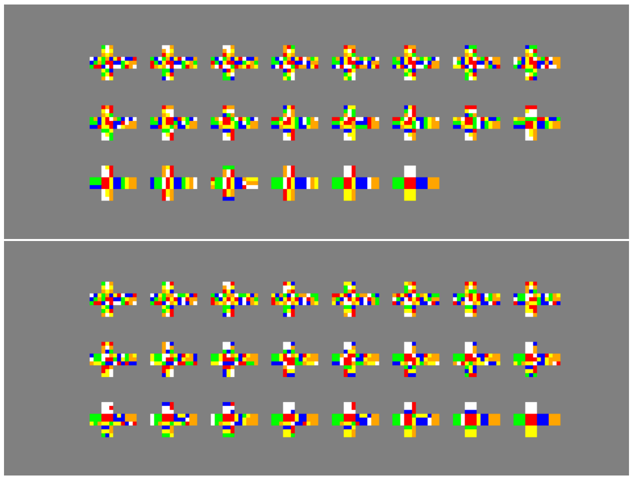

Figure 2. An example DCA solve trajectory (top) and DCA-LBL solve trajectory (bottom) for the same scramble. Solving of the first layer (white) is prioritized for a solve trajectory, providing visual incremental progress

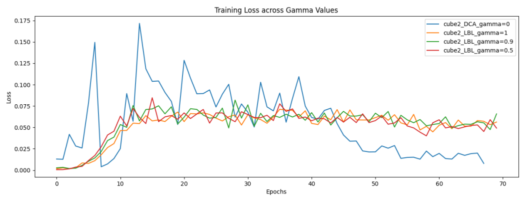

Figure 3. Corners-only (Cube2) Training Loss

trajectories that correctly positioned white pieces on the white face received a greater reward.

Our reward shaping framework for DCA-LBL shows that as γ increases, on average the white face is solved more quickly during a solve trajectory (Figure 4). This reveals that during the intermediate stages of a solve, our first layer tends to have more correctly positioned pieces, reflecting our LBL reward bias. This shows that a single layer tends to be solved before the completion of the entire cube all at once. As the slope for DCA-LBL trajectories is more constant over the course of an entire solve trajectory, this means that progress (% of white solved) is much more incremental and constant, thus visually perceptible and progress more human-interpretable.

In Table 1, we see that LBL reward-shaping not only solves the white face earlier during a trajectory but reinforces that the white layer stays solved over the course of an entire trajectory. Comparing DCA and DCA-LBL ($\gamma = 0.9$), we see first layer progress is not only achieved but substructure is also maintained during a solve trajectory. Figure 5 vi-

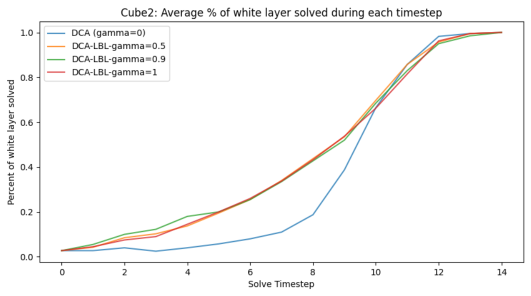

Figure 4. How quickly white is solved during a solve trajectory comparing DCA and DCA-LBL. This is over an average of 100 solves.

Table 1. Average % of white solved over an entire solve trajectory. Averaged over 100 solves.

γ	White % Solved	StDev
0	0.188	0.064
0.5	0.309	0.093
0.9	0.319	0.091
1	0.308	0.082

sualizes our corners-only approach with our layer-by-layer approach.

4.3. Corners-only + LBL: Tradeoffs and Affordances

Traditionally, there is a tradeoff in solving efficiency (fewest moves) when prioritizing a layer-by-layer approach to solving the Rubik's cube. Our DCA-LBL algorithm and reward-shaping framework tries to achieve a balance between optimality and interpretability, depending on our hyperparameter settings. As DCA-LBL optimizes for both solve efficiency and interpretability, we see that move efficiency is comparable to DCA even for more interpretable solutions. Figure 6 shows the average steps it takes A* to solve the 2x2 Rubik's Cube across γ values.

We were also curious about whether reward shaping helps during the solving process, specifically in reducing the number of nodes needed to be explored during the final A* search. Comparing across the spectrum of trading off shortest trajectories vs. interpretability, for $\gamma = 0.5, 0.9$, we see the average number of nodes searched dramatically decrease compared to the implementation with no reward shaping, DCA (Figure 7). This suggests that we may think about reward shaping as a form of regularization that trades variance in solve trajectory length for bias (less optimal trajectories). Indeed, since the algorithm is being incentivized to maintain the top layer throughout the solve, we expect fewer nodes may need to be explored since more of them are likely to disrupt the top node.

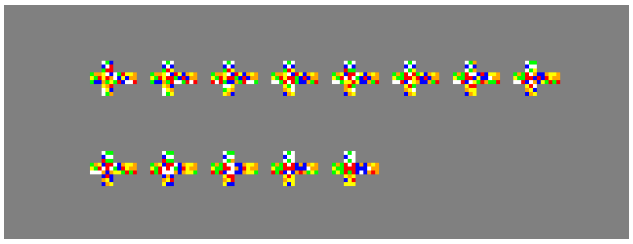

Figure 5. Solve trajectory visualization for Corner-only with Layer-by-Layer approach.

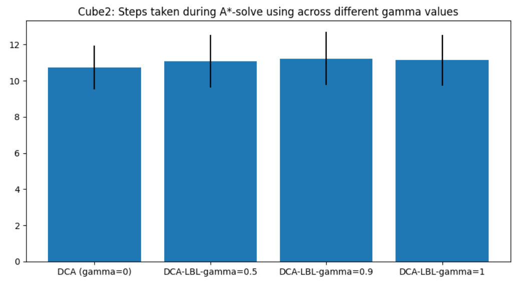

Figure 6. Solve trajectory length between γ values blending move efficiency and solve interpretability. Higher γ means that rewards are shaped to bias an LBL approach

However, for $\gamma = 1$, we found (somewhat surprisingly) that the number of nodes explored was significantly higher than both $\gamma = 0$ and $0 < \gamma < 1$. We argue that this is due to excessive reward shaping. At $\gamma = 1$, any move that fully preserves the top layer will accrue *zero* total transition cost, which means that any such move will be explored by A*. Since most of these moves only serve to further scramble the cube, this represents a huge waste in nodes explored and leads to unnecessarily longer runtimes when solving. The optimal value of γ then is likely somewhere between $\gamma = 0.9$ and $\gamma = 1$.

After generating the median of the number of nodes explored for the four algorithms, we find that DCA has a high node count mainly due to some solution trajectories that requires many more nodes visited before fully explored. Specifically, about 16% of the solution trajectory for DCA have over 3000 nodes explored (Figure 7). We think this suggests that DCA-LBL with a sensible Gamma setting also allows the searching process to be more stable across different starting scrambled states. Additional data related to this behavior is in Appendix 2.

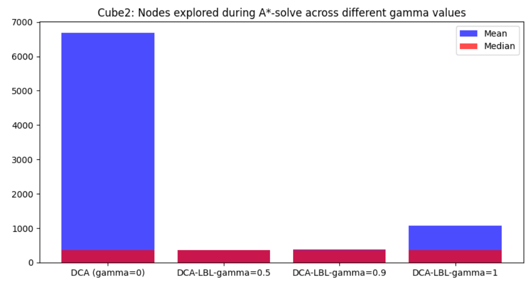

Figure 7. The number of nodes explored during A*-solve. Note that the mean and median for $\gamma = 0.5$ and 0.9 is similar. Higher γ means that rewards are shaped to bias an LBL approach. A more detailed histogram for each model is provided in the Appendix.

5. Conclusion

We enhance the interpretability of a DCA Rubik's Cube algorithm with human knowledge. Through reward shaping, we guide the training process of the DVN for value approximation within the algorithm using a human understanding of the process when solving the cube. This results in a searching algorithm that prioritizes creating and maintaining stable substructures on the cube during solution exploration, which is very similar to those a human would create when using methods such as layer-by-layer or corners-then-edges. Furthermore, we find that this training process allows the original DeepCubeA algorithm to find better value approximations of states, in the sense that using the new DVN as the heuristic function for an A* search allows the algorithm to explore less and more stabilized number of nodes across solution trajectories at almost no cost to the trajectory length on average. This shows that reward shaping inspired by human methods is an effective way of attaining a faster and more interpretable Rubik's Cube algorithm.

For future work, we would like to generalize our findings by exploring the effect of our algorithm over more scrambled states of the 3x3 Rubik's Cube. We also want to conduct experiments on shaping the reward function using other human methods. Finally, though adding subroutines to the action space resulted in computationally-intractable training for us in this project, we hope to explore new ways of taking advantage of human subroutines that reduce the state space and further enhance the interpretability of the resulting algorithm.

Contributions

Equal contribution for our proposal, milestone, report, and poster.

Cheng Chang Developed human subroutine model structure; created data and solve trajectory visualizations. Also solved a cube for the first time!

Jerry Liu Implemented intermediary reward-shaping, multi-stage solution-finding, trained models evaluated solve trajectories.

Terry Feng Implemented human subroutines action space and built corners-only model environment; provided data visualizations.

Acknowledgement

We would like to thank Professor Emma Brunskill and Skanda Vaidyanath for providing insights and suggestions for this work.

References

Agostinelli, F., McAleer, S., Shmakov, A., and Baldi, P. Solving the rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8): 356–363, Aug 2019. ISSN 2522-5839. doi: 10.1038/s42256-019-0070-z. URL https://doi.org/10.1038/s42256-019-0070-z.

Brunetto, R. and Trunda, O. Deep heuristic-learning in the rubik's cube domain: An experimental evaluation. In *Conference on Theory and Practice of Information Technologies*, 2017.

Kociemba, H. Two-phase algorithm details, 1992. URL http://kociemba.org/math/imptwophase.htm.

Korf, R. E. 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, pp. 700–705. AAAI Press, 1997. ISBN 0262510952.

McAleer, S., Agostinelli, F., Shmakov, A., and Baldi, P. Solving the rubik's cube without human knowledge, 2018. URL https://arxiv.org/abs/1805.07470.

Appendix

A. Github Repository

The GitHub repository containing codes related to our work is at https://github.com/thetrueconfusionist/DeepCubeA.

B. Additional Data

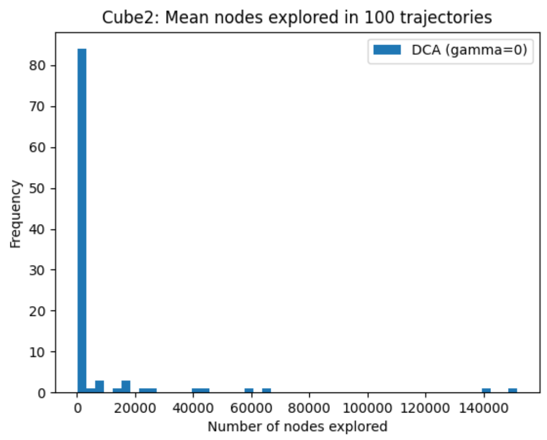

Figure 8. Mean number of nodes explored in 100 trajectories for the original DeepCubeA algorithm in the 2x2 cube environment.

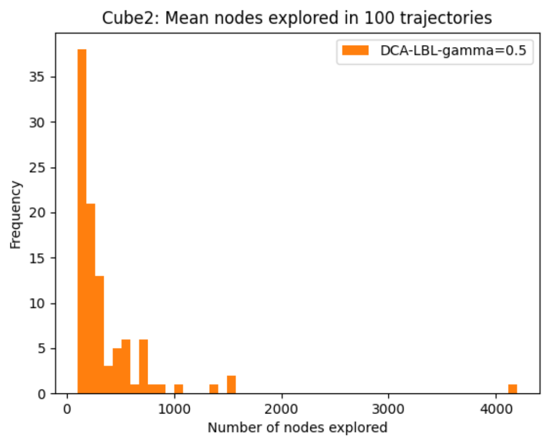

Figure 9. Mean number of nodes explored in 100 trajectories for our reward shaping algorithm (DCA-LBL) with $\gamma = 0.5$ in the 2x2 cube environment.

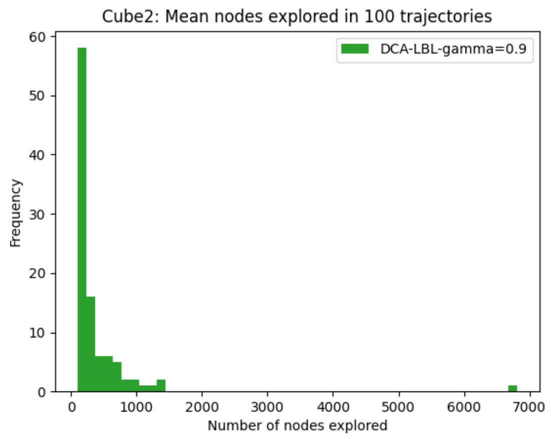

Figure 10. Mean number of nodes explored in 100 trajectories for our reward shaping algorithm (DCA-LBL) with $\gamma = 0.9$ in the 2x2 cube environment.

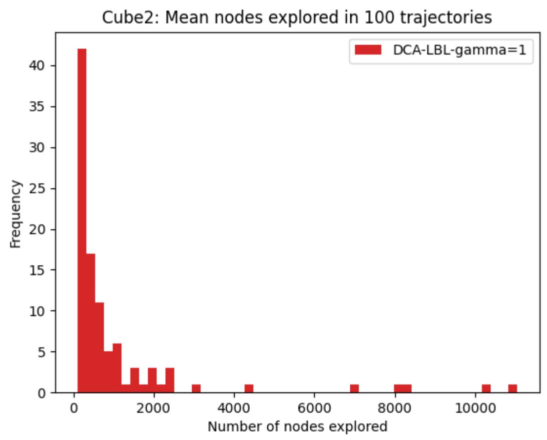

Figure 11. Mean number of nodes explored in 100 trajectories for our reward shaping algorithm (DCA-LBL) with $\gamma = 1$ in the 2x2 cube environment.

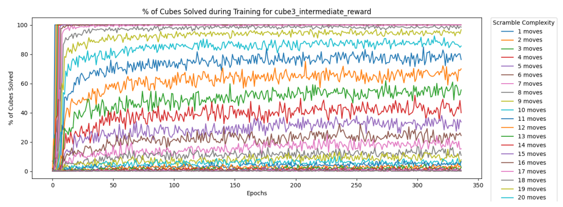

Figure 12. Cube 3 Training