# SELF-SUPERVISION IS ALL YOU NEED FOR SOLVING RUBIK'S CUBE

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

While combinatorial problems are of great academic and practical importance, previous approaches like explicit search and reinforcement learning have been complex and costly. To address this, we developed a simple and scalable method to train a Deep Neural Network (DNN) through self-supervised learning for solving a goal-predefined combinatorial problem. Assuming that more optimal moves occur more frequently as a path of random moves connecting two combinatorial states, we show that the DNN can produce near-optimal solutions by learning to predict the last move of a random scramble based on the problem state. We compare our method to an optimal solver and DeepCubeA (Agostinelli et al., 2019), by solving 1,000 Rubik's Cubes using the beam search. Although our model does not reach the DeepCubeA baseline, its performance scales up with more training samples and wider beam widths. The proposed method may apply to other goal-predefined combinatorial problems, though it has a few constraints.

## 1 INTRODUCTION

Combinatorial search is a challenging task both in theory and in practice. Representative problems include *Traveling Salesman Problem*, in which a salesman tries to travel through multiple cities in the shortest path possible. Despite how easy it sounds, the problem is labeled *NP-hard* due to its combinatorial complexity (Papadimitriou & Steiglitz, 1998); as the number of cities increases, the number of possible combinations skyrockets. In the real world, algorithms for such problems are applied in various ways to optimize logistics, supply chain, resource planning, etc.

In the past, researchers have proposed several shortcut algorithms to tackle the combinatorial complexity of such problems. Recently, in particular, reinforcement learning has enabled automatic search of algorithms to solve combinatorial problems near-optimally (Mazyavkina et al., 2021); by rewarding efficient moves, DNNs can learn to find increasingly more optimal solutions. *Rubik's Cube* is also one such problem (Demaine et al., 2018), and there have been not only logical search methods (Korf, 1997; Rokicki et al., 2014) but also heuristic (Korf, 1982; El-Sourani et al., 2010; Arfaee et al., 2011) and reinforcement learning approaches (McAleer et al., 2018; Agostinelli et al., 2019; Corli et al., 2021) that can solve any scrambled state of the puzzle. Nevertheless, combinatorial search is still no easy task; you need detailed knowledge of the target problem, coding explicit logic, or configuring a reinforcement learning system. Reinforcement learning is especially notable for the need to adjust architecture, loss function, and hyperparameters several times to stabilize the learning process often involving non-differentiable operations.

To overcome this difficulty, we developed a novel method to train DNNs in a self-supervised manner for solving combinatorial problems with a predefined goal. The method teaches a DNN probability distributions of optimal moves associated with problem states, with an *assumption* that, for a goal-predefined problem, random moves stemming from the goal state are probabilistically biased towards optimality. In our experiment, by training a DNN to solve Rubik's Cube, we evaluate our method against an optimal solver and DeepCubeA (Agostinelli et al., 2019).

**Contribution.** We show that self-supervised learning can be all you need for solving some goal-predefined combinatorial problems near-optimally, without requiring specialized knowledge or reinforcement learning.

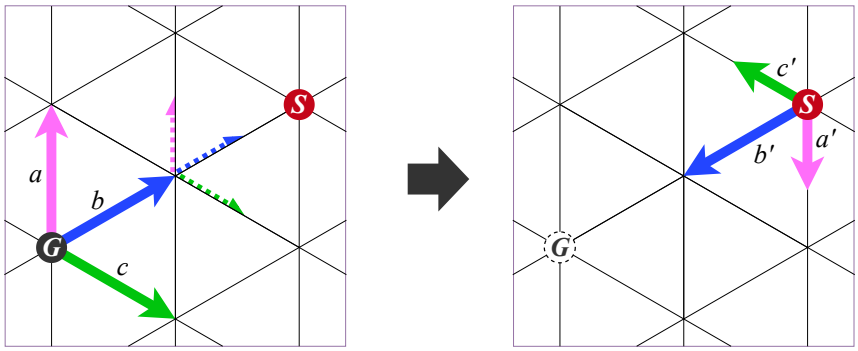


Figure 1: A simplified goal-predefined combinatorial problem in training and inference. When random moves are probabilistically biased toward optimal moves, a model learns this distribution and uses it for inference. $Left$: During training, random moves $\{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\}$ can occur at every node by the equal probability of $1/3$. If an unknown path consisting of these duplicable moves connects nodes $G$ (goal) and $S$ (scramble), the optimal (shortest) path $(\boldsymbol{b}, \boldsymbol{b})$ is most likely of all the paths possible, followed by the second optimal three-move paths and then four-move paths. At all nodes, adding up the probabilities of all the possible paths, a move in more optimal paths turns out to have a higher probability of being in the unknown path. Thus, the model can learn the probability distributions merely by observing a sufficient number of random moves derived from $G$. $Right$: Using the learned probability distributions, the model can infer reverse paths from node $S$ to the unknown goal $G$ consisting of the inverse moves $\{\boldsymbol{a'}, \boldsymbol{b'}, \boldsymbol{c'}\}$. Since the moves in the shortest path are learned most frequent of all paths, in this instance, the move $\boldsymbol{b'}$ is predicted to be more optimal than $\boldsymbol{a'}$ and $\boldsymbol{c'}$ at node $S$ and its subsequent node.

## 2 Proposed Method

In this section, we first provide an overview of the proposed method and then discuss its constraints. Below, we consider a combinatorial problem as a pathfinding task on an *implicit* graph. Each node in the graph represents a unique state and has edges as moves to its adjacent nodes with an equal probability.

### 2.1 Overview

The fundamental idea of the proposed method is as follows: 1) apply a sequence of random moves to the target problem, and 2) train a DNN to predict the last random move based on the problem state. By learning associations between problem states and probability distributions of the last moves, the DNN can infer step-by-step moves as a reverse path to the initial goal state. After training, the multinomial probability distribution of moves at inference step $i$ should approximate

$$P(X_i = j : j \in \mathbb{M}) = \sum_{k \in \mathbb{M}} P(X_{i+1} = k) \cdot P(X_i = j \mid X_{i+1} = k) \tag{1}$$

where $X_i$ is a choice of move at step $i$, and $j$ and $k$ belong to the set of all available moves $\mathbb{M}$ at steps $i$ and $i+1$. Here, the choice at step $i$ does not *actually* depend on the future step $i+1$. Rather, it should *approximate* how similar states depended on past steps during training, which in turn can be viewed as future steps at inference. In this sense, the probability distribution of the move $X_i$ *implicitly* depends on paths to the current state or similar that appeared during training. Figure 1 shows a miniature example.

### 2.2 Condition and Assumptions

There is one necessary condition for the proposed method to work, one assumption for a DNN to approximate towards an optimal solver, and another assumption for effective training and inference. First, to be solved with the proposed method, a problem *must* meet the following condition.

**Condition 1.** For the problem of interest, its goal state is predefined and readily available.

For a DNN to learn probability distributions of moves connecting to the goal state, any path consisting of random moves must be derived from the goal state. Therefore, Condition 1 is necessary for the proposed method.

Subsequently, the following two assumptions are *desirable* for a self-supervised DNN to approximate an optimal solver.

**Assumption 1.** The more optimal a move is, the more frequently it occurs.

Although Assumption 1 is not necessary for applying the proposed method, the more true Assumption 1 is for the problem, the more optimal the DNN will be. Here, we formulate the statistical condition for Assumption 1 to be true on a problem when all moves at every node have an equal likelihood. Let $p_{n,a}$ be the probability of taking a particular move $a$ as the first step to a specific target node in a path of $n$ moves or more,

$$p_{n,a} = \sum_{k=n}^{k_{max}} \frac{C_{k|n}}{A^k} \tag{2}$$

where $k$ is the move count to the target state, $k_{max}$ is the maximum number of $k$, $A$ is the total number of possible moves at all nodes, and $C_{k|n}$ is the number of $k$-move paths ($k \geq n$). Likewise, let $b$ be an alternative to $a$ that requires $n+1$ moves or more, Assumption 1 is true if and only if $p_{n,a} \geq p_{n+1,b}$. Thus, provided $p_{n,a} = C_{n|n}/A^n + \sum_{n+1}^{k_{max}}(C_{k|n}/A^k)$, the following must be satisfied for any $n$ for Assumption 1 to be true:

$$\frac{C_{n|n}}{A^n} + \sum_{k=n+1}^{k_{max}} \frac{C_{k|n}}{A^k} \geq \sum_{k=n+1}^{k_{max}} \frac{C_{k|n+1}}{A^k} \tag{3}$$

At this point, both $C_{k|n}$ and $C_{k|n+1}$ are unknown on an implicit graph, and the $\sum$ terms do not necessarily equal when the paths start with different moves. However, the only potential difference is the first move in two paths of different optimalities, and just taking a sub-optimal $b$ move should not largely increase the total number of same-distance paths. Therefore, we assume that this holds true to varying degrees depending on the problem being addressed.

**Assumption 2.** The maximum number of moves required to solve any state—so-called *God's Number*—or at least its upper bound should be known for a target problem.

In the proposed method, let $N_G$ be God's number of a given problem, the DNN receives problem states shuffled for $1 \sim N_G$ moves. If you estimate the number smaller than it actually is, then the DNN would not well adapt to problem states of higher complexities during inference. On the contrary, if you overestimate $N_G$, you may unnecessarily apply additional moves to states of the highest complexity, resulting in more non-optimal moves in training scrambles. One way to address this limitation would be to keep increasing the maximum number of shuffles, as long as the DNN can make predictions beyond chance for the maximum complexity.

If Condition 1 and Assumption 1 are met, random moves should have expected probability distributions peaking around optimal moves. It is also notable that the degree to which a problem satisfies Assumption 1 affects the optimality of the trained DNN: the more possible paths violating Assumption 1, the less optimal the DNN will be. Although the example in Figure 1 has a very restricted setting for explanatory simplicity, this observation could apply to similar problems of higher dimensions and complexities under these constraints. Accordingly, we hypothesize DNNs to learn to solve goal-predefined combinatorial problems near-optimally.

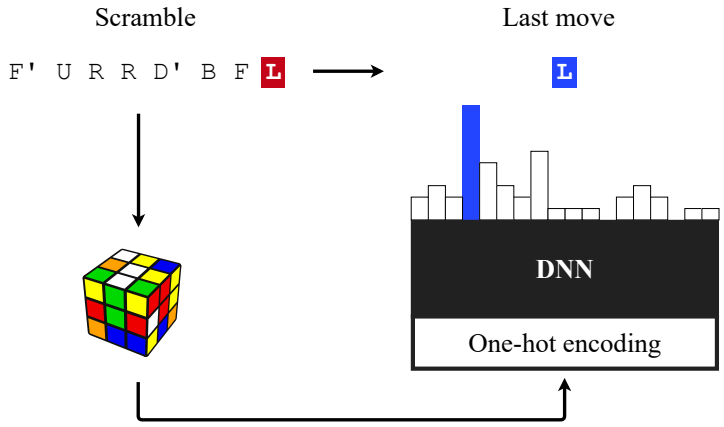


Figure 2: An illustrated scheme of the proposed self-supervised learning on Rubik's Cube. Applying a sequence of random moves [F' U R R D' B F L] to Rubik's Cube, the DNN receives the one-hot representation of the scrambled state as input and predicts the last move in the scramble.

# 3 Experiment

## 3.1 Rubik's Cube

Rubik's Cube is a cubic puzzle with 6 faces each having 9 color stickers. The goal of the puzzle is to move the stickers by rotating pieces on 6 faces, so that each face has stickers of only one color. Including the goal state, the puzzle can have approximately $4.3 \times 10^{19}$ different states.

In our experiment, the puzzle is represented by its sticker locations and their colors. By assigning indices to all sticker locations and colors, one-hot encoding can represent the puzzle in 324 dimensions (54 sticker locations$\times$6 colors). For rotating Rubik's Cube, we employ the quarter-turn metric (a 90° turn of a face counts as one move whereas a 180° turn counts as two), meaning 12 possible moves for every given state. In this configuration, solving Rubik's Cube can be described as an implicit graph search problem in the 12-$D$ action space.

## 3.2 Learning Objective

Given the state of a Rubik's Cube scrambled for random $1 \sim 26$ moves[1], a DNN predicts the last move of a scramble (see Figure 2). Here, we set the maximum scramble length to 26 because the number is known as sufficient for optimally solving any state of Rubik's Cube (Kunkle & Cooperman, 2007). Since the inverse of the last scramble move is expectedly close to optimal under Assumption 1, we hypothesize that the DNN can learn to solve the puzzle near-optimally through this learning objective.

## 3.3 Model

We employ the same architecture as DeepCube and DeepCubeA (McAleer et al., 2018; Agostinelli et al., 2019). The model first has two fully-connected (FC) layers, followed by four residual blocks (He et al., 2016) each containing two FC layers. Finally, for the prediction output, the model has a 12-$D$ FC layer with softmax activation.

The DNN is trained for $1,000,000$ steps with a fixed batch size of $5,200$ (200 states per scramble length $\times$ 26 scramble moves). We use Adam optimizer (Kingma & Ba, 2014) with the initial learning rate of $10^{-4}$ for updating model weights.

[1]We exclude obviously redundant moves like R (90° clockwise turn on the right face) following R' (counterclockwise).

### 3.4 INFERENCE

For solving Rubik's Cube with the trained model, we adopt a simple beam search. Starting from depth 1 with a scrambled state, at every depth $i$, the DNN predicts the next moves for every candidate state. Let $k$ be the width of the beam search, we pass at most $k$ candidate paths and corresponding states to the next depth $i+1$, sorted by their expected probabilities. Also, for every potential move, if a move $a$ is predicted to be worse than chance (i.e., $p(X_i = a) < 1/12$), we exclude the path from the set of candidates before sorting. The search continues until any of the candidate states is solved, at which point the search depth matches the solution length.

## 4 RESULTS

To test our method on Rubik's Cube, we use the same dataset as DeepCubeA (Agostinelli et al., 2019), which contains $1,000$ Rubik's Cubes randomly scrambled for $1,000 \sim 10,000$ moves[2]. We evaluate checkpoint models trained for $0.25$, $0.5$, and $1.0$ million steps (0.25M, 0.5M, and 1.0M models) by solving the test cases with beam widths scaling in factorials of 2 from $2^8$ to $2^{15}$, and compare the results to those of DeepCubeA and an optimal solver (Agostinelli et al., 2019).

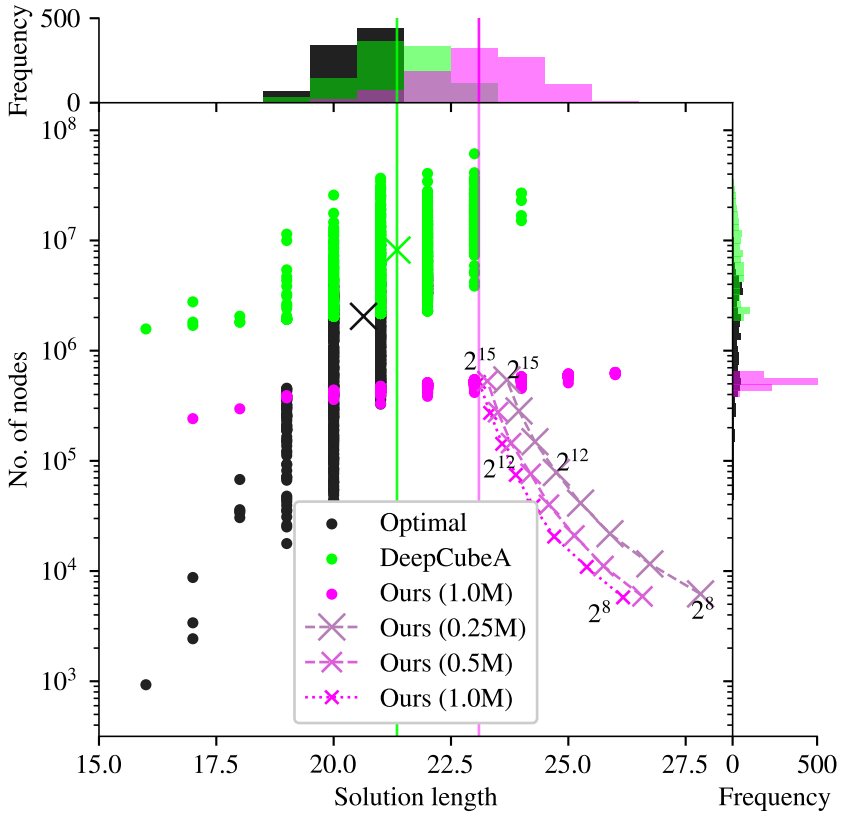


Figure 3: Solution length versus number of nodes by three methods: Optimal solver, DeepCubeA, and ours. Each dot represents the number of moves in a solution and the number of nodes expanded during a search. For a clear comparison, we scatter plot the result of solutions by 1.0M model with the beam width of $2^{15}$ only. Frequency plots are separately shown for both dimensions, and The pink dashed line shows the trajectory of mean coordinates (cross markers) scaling up along the beam widths for each model. Some crosses are annotated by their corresponding beam widths.

[2]Code Ocean Capsule by Agostinelli et al. (2019) (doi.org/10.24433/CO.4958495.v1)

With the beam widths of $2^8 \sim 2^{15}$, all checkpoint models successfully solved all test cases. Figure 3 shows how our result compares to the results of DeepCubeA and the optimal solver[3] in terms of solution length and number of nodes expanded during solution search.

Similarly, Figure 4 plots the amount of time taken and the solution length to solve for every test case. Because the calculation time strongly depends on environmental configurations (e.g., number of GPUs, distributed processing, etc.), we normalized our results so that the average per-node computation time matches that of DeepCubeA. By doing so, we virtually simulate the temporal performance of our model in a DeepCubeA-equivalent environment. As Figure 5 shows, there are strong correlations between number of nodes and calculation time.

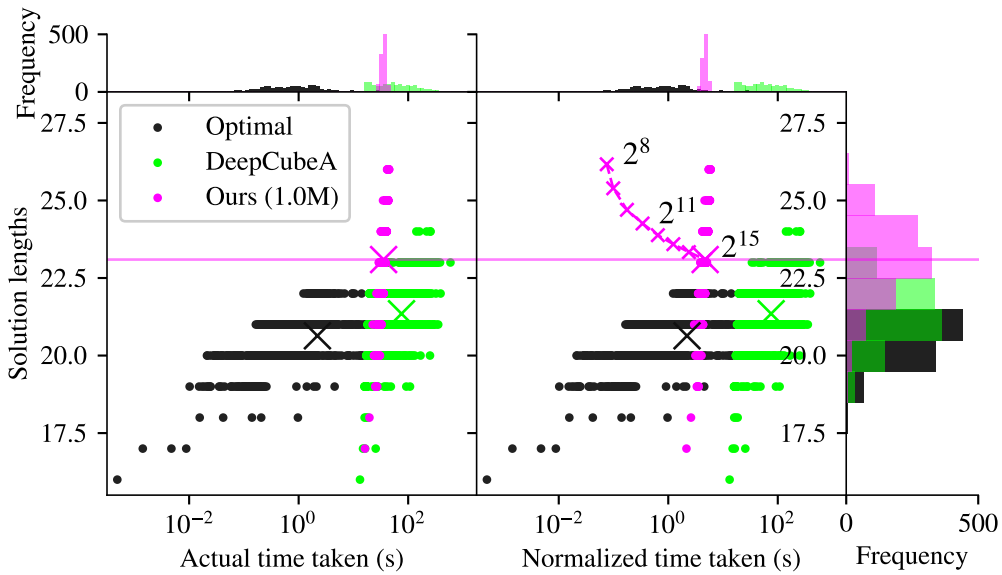


Figure 4: Calculation time versus solution length by three methods. The scatterplot on the left shows the actual time taken, and the right one shows the time normalized by per-node computation time. Similar to Figure 3, the mean coordinates are marked by a cross for all results.

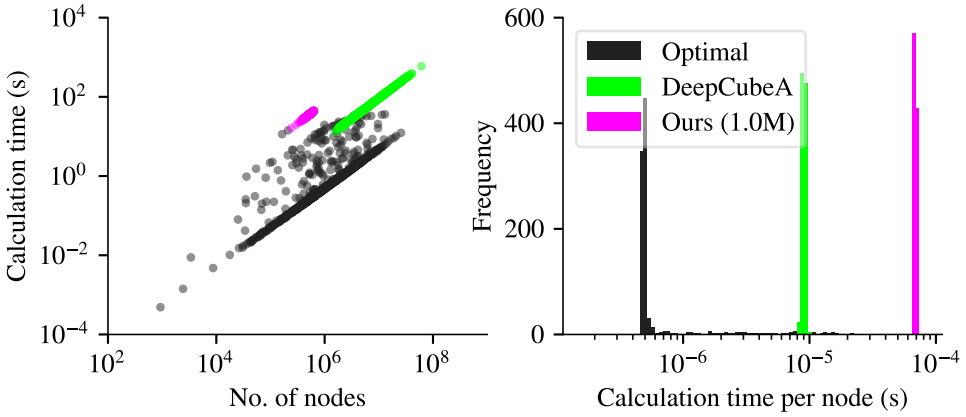


Figure 5: Associations between the number of nodes and calculation time for three different methods. *Left*: Strong correlations between the two metrics in log scales. *Right*: Distributions of computation time per node for the optimal solver, DeepCubeA, and ours (1.0M model).

[3]Solutions provided on GitHub repository (`github.com/forestagostinelli/DeepCubeA/`)

Table 1: Different methods' performance on solving Rubik's Cube. Mean values are reported for solution length, number of nodes, and time taken to solve per test scramble. Optimal (%) indicates the percentage of optimal solutions. The last row shows the calculation time normalized in comparison to DeepCubeA as in Figure 4.

| Method | Solution length | No. of nodes | Time taken (s) | Optimal (%) |
|---|---|---|---|---|
| Optimal solver | 20.64 | $2.05 \times 10^6$ | 2.20 | 100.0 |
| DeepCubeA (paper) | 21.50 | $6.62 \times 10^6$ | 24.22 | 60.3 |
| DeepCubeA (GitHub) | 21.35 | $8.19 \times 10^6$ | 75.61 | 65.0 |
| **Ours** | 23.09 | $0.52 \times 10^6$ | 35.60 | 12.6 |
| Ours (Normalized) | | | 4.72 | |

Table 1 summarizes the statistics of different methods, including the paper result of DeepCubeA (Agostinelli et al., 2019). For our method, we report the result for solutions by 1.0M model with the beam width of $2^{15}$.

## 5 DISCUSSION

We proposed an intuitive and easy-to-implement method to solve goal-predefined combinatorial problems. Self-supervised on random moves and corresponding problem states, our DNN successfully learned to solve Rubik's Cube. Still, this success does not indicate the degree to which Rubik's Cube satisfies Assumption 1 as an implicit graph search problem.

Although our method did not outperform DeepCubeA in terms of solution optimality, it appears to be scalable and still has the prospect of reaching the baseline. As Table 1 shows, while DeepCubeA generated $650$ optimal solutions for $1,000$ test cases, our model did so for only $126$ cases in the experiment. Also, when optimal solutions average $20.64$ moves, our model generated solutions with an average length of $23.09$, whereas DeepCubeA solutions are $21.35$ moves long on average. However, we believe that this difference is largely attributed to the amount of computation. Whereas DeepCubeA was trained on $10$ billion examples (Agostinelli et al., 2019), our model was trained only on the equivalent of $0.2$ billion examples (1 million steps $\times 200$ per scramble length). Also, while searching for solutions, DeepCubeA expanded an average of $8.19 \times 10^6$ nodes, while ours did only $0.52 \times 10^6$ nodes per case. In addition to this, as Figure 3 shows, our method is scalable; the inference performance scales up with the number of training samples and the beam widths. Therefore, our method might still reach the DeepCubeA baseline, and possibly the optimal solver as well, with more training and inference computation.

Also, because of the strong correlations between number of nodes and calculation time (Figure 5), our method has a trade-off between calculation time and solution optimality, as Figure 4 shows. For a practical application, one might control the beam width so that the DNN can find good enough solutions in a reasonable time, depending on the execution environment. Like language models (Kaplan et al., 2020), enlarging the model might also contribute to the solution performance. Future research could study the scaling laws for higher efficiencies and performance, making the best use of limited time and computational resources.

As the most important limitation, when applying the proposed method, the target problem must have a predefined goal (Condition 1). Therefore, our method cannot apply to multiplayer games like chess, in which no goal state is available in advance, whereas reinforcement learning is an appropriate and effective approach (Silver et al., 2018). Likewise, problems like Traveling Salesman Problem, whose objective is to find the goal combinatorial state itself, will not be best solved by our method.

Nonetheless, the proposed method has potential applications to other problems. Our method would also be suitable for other goal-predefined puzzles like 15 puzzle, Light Out, Sokoban, etc., which are also solved by DeepCubeA (Agostinelli et al., 2019). We show an application of our method on 15 puzzle in Appendix A. If Assumption 1 is met, applications might extend to real-world problems like route planning with variable-distance moves, for example, by weighting probability distributions of random training moves inversely by move distances. Another potential application, which we did

not examine in this work, is to problems where the action space is continuous. For instance, if a robot has the task of pressing a button with its finger, starting with the finger touching the button (goal state), we could apply the inverse of a random move based on continuous probability distributions of motors for training. To the best of our knowledge, such a training method has not been reported.

Our method could also be used as pre-training for further optimization with reinforcement learning. The *policy* a DNN learns in our method is *blurred* due to the randomness of training moves, which reinforcement learning could potentially rectify. This pretraining would provide stable guidance for an agent learning the optimal policy.

In the specific domain of Rubik's Cube, it may well adapt to any algorithmic demands and constraints. For example, if you want to develop a robotic Rubik's Cube solver with only five motors, you only need to train/finetune a model without rotating a specific face in random scrambles. For another, you may tune the method for ergonomic optimality. Controlling the probability distribution of random moves in training scrambles may result in easy-to-execute solutions at inference.

## 6 Conclusion

In this work, we introduced a simple and scalable method to solve goal-predefined combinatorial problems through self-supervision. Our method trains a DNN to predict the last move of a random scramble given a problem state, with an assumption that mere random moves can form probability distributions biased towards optimalities associated with problem states. Tested on Rubik's Cube, the trained DNN successfully solved all the $1,000$ test cases near-optimally. Though with some constraints, the proposed method is potentially applicable to similar combinatorial problems as well, possibly with fewer constraints. It may also provide new ideas for self-supervised learning and reinforcement learning.

## Reproducibility Statement

We provide a jupyter notebook for reproducibility testing:

`colab.research.google.com/drive/1EOFrB-LxgW0AeRzrFNTpTupqyDbxUjEp`

## References

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), 2019.

Shahab Jabbari Arfaee, Sandra Zilles, and Robert C Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17), 2011.

Adrian Brüngger, Ambros Marzetta, Komei Fukuda, and Jürg Nievergelt. The parallel search bench zram and its applications. *Annals of Operations Research*, 90:45–63, 1999.

Sebastiano Corli, Lorenzo Moro, Davide Galli, and Enrico Prati. Solving rubik's cube via quantum mechanics and deep reinforcement learning. *Journal of Physics A: Mathematical and Theoretical*, 2021.

Erik D Demaine, Sarah Eisenstat, and Mikhail Rudoy. Solving the rubik's cube optimally is np-complete. *arXiv:1706.06708v2*, 2018.

Nail El-Sourani, Sascha Hauke, and Markus Borschbach. An evolutionary approach for solving the rubik's cube incorporating exact methods. *European Conference on the Applications of Evolutionary Computation*, pp. 80–89, 2010.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv:2001.08361*, 2020.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980*, 2014.

Richard E Korf. A program that learns to solve rubik's cube. *AAAI*, pp. 164–167, 1982.

Richard E Korf. Finding optimal solutions to rubik's cube using pattern databases. *AAAI/IAAI*, 1997.

Richard E Korf. Linear-time disk-based implicit graph search. *Journal of the ACM (JACM)*, 55(6): 1–40, 2008.

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*, pp. 235–242, 2007.

Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey. *Computers & Operations Research*, 2021.

Stephen McAleer, Forest Agostinelli, Alexander Shmakov, and Pierre Baldi. Solving the rubik's cube with approximate policy iteration. *International Conference on Learning Representations*, 2018.

Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Courier Corporation, 1998.

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

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419), 2018.

## A 15 PUZZLE

We present one application of the proposed method other than Rubik's Cube: *15 puzzle*. 15 puzzle is a sliding puzzle consisting of 15 numbered tiles and 1 empty slot on a 4×4 board. The goal is to align the tiles in order of their numbers, by repeatedly swapping the empty slot with neighboring tiles. Including the goal, the puzzle can take approximately $1.0 \times 10^{13}$ states, and it is known to be solved in 80 moves or fewer (Brüngger et al., 1999; Korf, 2008).

Like for Rubik's Cube, we trained a single model on $8,000,000$ samples ($100,000$ steps $\times 10$ samples per shuffle length) and tested on the same DeepCubeA dataset (Agostinelli et al., 2019) containing 500 test cases for this puzzle. Using the beam search, the trained model successfully solved all the cases with beam widths of $2^7 \sim 2^{12}$. Like for Rubik's Cube, the mean solution length decreases with wider beam widths (see Figure 6). Table 2 summarizes the different results. For this puzzle, we do not control for per-node computation times.

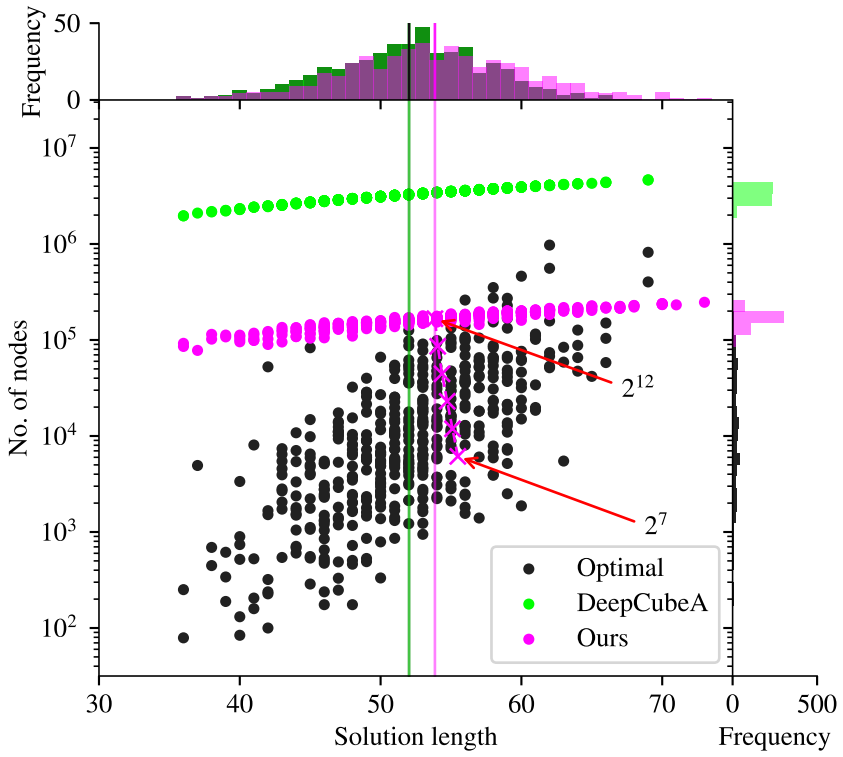


Figure 6: Solution length versus number of nodes generated during solution search.

Table 2: Different methods' performance on solving 15 puzzle. Mean values are reported for solution length, number of nodes, and time taken to solve per test scramble. Optimal (%) indicates the percentage of optimal solutions.

| Method | Solution length | No. of nodes | Time taken (s) | Optimal (%) |
|---|---|---|---|---|
| Optimal solver | 52.02 | $3.22 \times 10^4$ | 0.0019 | 100.0 |
| DeepCubeA (Paper) | 52.03 | $3.85 \times 10^6$ | 10.28 | 99.4 |
| DeepCubeA (GitHub) | 52.02 | $3.28 \times 10^6$ | 8.82 | 100.0 |
| **Ours** | 53.86 | $0.17 \times 10^6$ | 54.44 | 39.4 |