Solving Twisty Puzzles Using Parallel Q-learning

Kavish Hukmani, Sucheta Kolekar*, *Member, IAENG,* Sreekumar Vobugari

***Abstract*—There has been a recent trend of teaching agents to solve puzzles and play games using Deep Reinforcement Learning (DRL) which was brought by the success of AlphaGo. While this method has given some truly groundbreaking results and it is very computationally intensive. This paper evaluates the feasibility of solving Combinatorial Optimization Problems such as Twisty Puzzles using Parallel Q-Learning (PQL). We propose a method using Constant Share-Reinforcement Learning (CS-RL) as a more resource optimized approach and measure the impact of sub- goals built using human knowledge. We attempt to solve three puzzles, the 2x2x2 Pocket Rubik's Cube, the Skewb and the Pyraminx with and without sub-goals based on popular solving methods used by humans and compare their results. Our agents are able to solve these puzzles with a 100% success rate by just a few hours of training, much better than previous DRL based agents that require large computational time. Further, the proposed approach is compared with Deep Learning based solution for 2x2x2 Rubik's Cube and observed higher success rate.**

***Index Terms*—Parallel Programming ; Q-learning ; Reinforcement Learning ; Twisty Puzzles ; Rubik's Cube ; Agent-based Programming**

I. Introduction

A twisty puzzle is a 3D puzzle made up of a set of pieces that can be arranged in a large number of states using a small number of actions in the form of twists. The Rubik's Cube is the most popular twisty puzzle and has over $4.3x10^{19}$ possible states which can be manipulated through a combination of six fundamental actions. Performing the same action on a particular state always gives the same result. This property allows us to represent the puzzles as Markov Decision Processes (MDPs), thereby making them suitable to be solved via Reinforcement Learning (RL). Reinforcement Learning has lot of applications domains to improve the performance of the system [1] [2] [3]. There have been a lot of advancements in the field of Deep Reinforcement Learning (DRL) recently which can be seen through the success of AlphaGo[4] and OpenAI Five[5]. These DRL models require large amounts of computation power and time to achieve good results. The aim of this paper is to reduce the computation by providing the agent with some human knowledge and using a more traditional RL approach. This is done by using sub-goals to reward certain intermediate states based on various methods used by humans to solve these puzzles.

Manuscript received June 22, 2021; revised October 27, 2021.

Kavish Hukmani, Student, Dept. of Information and Communication Technology, Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal, Karantaka, India, 576104, e-mail: (khukmani@gmail.com)

Sucheta Kolekar, Assistant Professor, Dept. of Information and Communication Technology, Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal, Karantaka, India, 576104, e-mail: (kolekar.sucheta@gmail.com)

Sreekumar Vobugari, Group Manager, Infosys Limited, Hyderabad, India, 500088, email (tcs.vobugari@gmail.com)

*Corresponding Author email: kolekar.sucheta@gmail.com

There is an ever growing community of twisty puzzle solvers worldwide. The World Cubing Association (WCA) conducts hundreds of speed-cubing events all around the world every year. This competition and inquisitiveness has lead to the creation of various methods to solve these puzzles, each with their own advantages and unique approaches. We aim to use these methods to create sub-goals for the agent to help it learn about these puzzle quicker. We compare a few common methods for each of the puzzles used.

The results of these experiments will help in determine the viability of using sub-goals in combination with CS-RL to emulate human behaviour for similar highly sequential problems which can be represented by MDPs. Some of these highly sequential problems are Supply Chain Optimization (SCO), DNA folding and Vehicle Routing.

The particulars of the experiment can be found in three sections; Twisty Puzzles (Section III), Methodology (Section IV) and Results (Section V). The twisty puzzles section contains background information about each of the puzzles including their structure and complexity. The methodology section has details about the environment used. It also contains information about the techniques and algorithms used to train the agents. Lastly, it describes the testing methodology used. The results section consists of various graphs and tables of multiple parameters that are used to measure the success of the agents. It also explains various trends seen and discusses possible reasons for the same. These results are summarized as a conclusion in Section VI along with the scope of similar experiments in the future.

II. Related Work

This section describes some of the contributions and research which have led to the solve twisty puzzles using Parallel Q-Learning. AlphaGo Zero [4] started a boom in the usage of DRL as a means to solve puzzles and play games. It used a combination of MCTS and self-play RL. It was a major breakthrough and a variant of the same system defeated the reigning Go World Champion Lee Sedol. It was trained using sixty-four GPU, nineteen and four TPU workers and servers for inference which costed approximately $25 Million[6].

Another popular breakthrough was the OpenAI Five [5], a collection of five individual agents that defeated reigning champions in a 5v5 game of Dota 2. It is one of the most popular games on Steam, a popular PC game store[7]. The Five used a scaled up version of Proximal Policy Optimization(PPO) in combination with a separate LSTM for each hero in the game[8]. It was trained on 256 GPUs and 128,000 CPU cores for a total of 180 years of gameplay[8].

DeepMind's AlphaStar [9] achieved a similar feat by becoming better at StarCraft II than 99.8% of players. The AI also beat various pro players in the 1v1 game mode. It was trained using a combination of supervised learning on

human player data and multi-agent RL. It was trained on 16 v3 TPUs for 14 days, giving each agent around 200 years of gameplay. DeepMind also created Agent57[10], a DRL agent that can play 57 Atari 2600 games and performs better than humans in all of them.

DeepCube [11] was an agent with a similar architecture to AlphaGo Zero. It used Autodidactic Iteration (ADI) to solve a 3x3x3 Rubik's Cube. It was trained using three Nvidia TitanXP GPUs paired with a 32-core Intel Xeon E5-2620 and required 44 hours to be trained.

A similar study, *Karmakar 2020*[12] adapted DeepCube for a 2x2x2 Pocket Rubik's Cube. It used a 2 core Intel Xeon @ 2.20GHz with 1 Nvidia Tesla K80 GPU for training. It was only able to achieve a 75% success rate when it required four or more moves to solve the puzzle.

As it can be seen, all these agents require large amounts of compute power and time to train. They each have different architectures and some like Agent57 are given rewards for meeting various sub-goals. The aim of the paper is to solve twisty puzzles such as the 3x3x3 Rubik's Cube but due to a lack of accessibility of similar high end hardware, we use some simpler puzzles instead such as the 2x2x2 Pocket Rubik's Cube. Another workaround is to use to implement the agents using PQL[13] instead of a NN based approach.

There are various algorithms that are classified as PQL, most of them either divide the states into groups for each agent[14] or are decentralized and have limited communication[15]. CS-RL, which is used in this experiment, has been mentioned theoretically[16], but no public usage or implementations of it exist. *Takamatsu 2007*[17] uses a similar approach but distinguishes between exploration and exploitation agents and combines the Q-tables periodically.

As there are no guidelines on the usage of sub-goals in solving puzzles using RL, we train our agent both with and without sub-goals and compare the results.

III. Twisty Puzzles

This section provides information on the workings of the twisty puzzles used in this experiment and how they are solved. This is used to create reward methods for the environment. It also outlines some basics of Q-learning and the OpenAI Gym. The following puzzles were chosen to be solved by the agents using various methods.

A. Pocket Rubik's Cube

The Pocket Rubik's Cube is a 2x2x2 variant of the Rubik's Cube. As seen in Figure 1, the puzzle is built from eight smaller corner pieces with three different colors. These make up six faces along the three axes with four tiles each. Each face can rotate 90 degrees in either direction, which rolls the tiles along the side of that face along with rotating the face, similar to the Rubik's Cube. It has 3,674,160 [18] possible combinations and requires a maximum of 11 moves to solve (also known as God's Number[19]).

1) Layer By Layer Method: The Layer By Layer (LBL) Method, also known as the Beginners Method, is the most popular method for solving a 2x2x2 Rubik's Cube. It is a scaled down version of the LBL method for the 3x3x3 Rubik's Cube. It consists of the following steps:

1) Solve the first layer
2) Orienting the Last Layer (OLL)
3) Permuting the Last Layer (PLL)

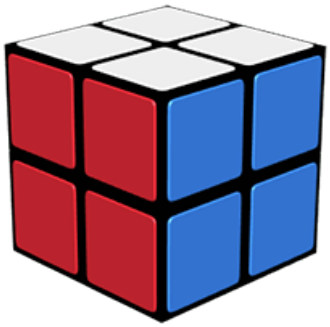


Fig. 1. Visualization of a Pocket Rubik's Cube (Source: grubiks.com)

The reason for choosing this method is its popularity and simplicity of steps. It requires only four algorithms to be solved, two for OLL and two for PLL. The first layer is solved intuitively.

2) Varasano/Ortega Method: The Varasano method, more commonly known as the Ortega Method, is a more advanced method for solving a 2x2x2 Rubik's Cube. It results in much quicker solves and is popular among speed cubers. It consists of the following steps:

1) Solve the face pieces of the first layer
2) Orienting the Last Layer (OLL)
3) Permuting Both Layers (PBL)

While the second step is the same in both the methods, in the Varasano/Ortega Method we only orient the first layer. The final step is permuting the two layers simultaneously via PBL. This consists of six possible cases, making the number of algorithms required eight.

B. Skewb

The Skewb is a corner turning puzzle with six faces as seen in Figure 2. It consists of eight corner pieces with three different colors each and six center pieces with a single color. It has four axes along the main diagonals which allow each corner to turn 120 degrees in either direction. Each move causes the respective center pieces and neighbouring corner pieces to also roll around the corner. It has 3,149,280 [18] possible combinations and a God's Number of 11.

1) : Sarah's Method Sarah's Method is the most popular method for solving a Skewb. It has three variants— Beginners, Intermediate and Advanced. We use the advanced variant which consists of the following steps:

1) Solve the first layer
2) Solve the Corners of the Last Layer (CLL) + remaining centers

This variant has a total of 134 cases for the second step. These cases can be broken down into sub cases resulting in the Beginner's and Intermediate variants. The first layer is solved intuitively.

C. Pyraminx

As seen in Figure 3, the Pyraminx is a tetrahedral puzzle with four equilateral triangles as faces. It is made up of four

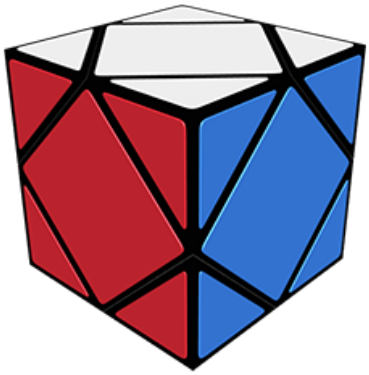


Fig. 2. Visualization of a Skewb (Source: grubiks.com)

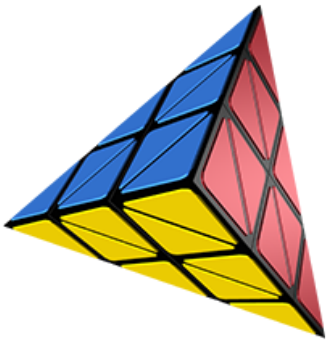


Fig. 3. Visualization of a Pyraminx (Source: grubiks.com)

tips with three colors on each, six edge pieces with two colors and four central pieces with three colors each for a total of fourteen pieces. It consists of four axes which each have a fixed central piece and a tip. There are two kinds of moves— rotating the tip and rotating the central piece. Both these moves are 120 degree turns around the axis and are independent of each other. Rotating the central piece also rolls the three edges around it.

As these tips are independent and require a maximum of one move to solve, we shall not consider them in order to reduce the number of combinations. The Pyraminx has 933,120 [18] possible combinations (excluding the tips) and a God's Number of 11.

1) Layer By Layer Method: The LBL Method for the Pyraminx is an Intermediate method for solving the puzzle. It's steps are similar to LBL Methods for other puzzles. They are:

1) Permuting the tips (This step can also be carried out last as the tips are independent pieces)
2) Solve the first layer
3) Solve the last layer

There are five cases for the last layer. This can be further broken down into two cases and a parity case. The first layer is solved intuitively.

D. Notation

The notation used to denote the moves is defined by the World Cubing Association(WCA) Regulations and Guidelines[20].

IV. Methodology

The following steps are followed in the Methodology and Experimentation:

- Creating an environment
- Creating a program to train agents using PQL
- Testing the results

A. Environment

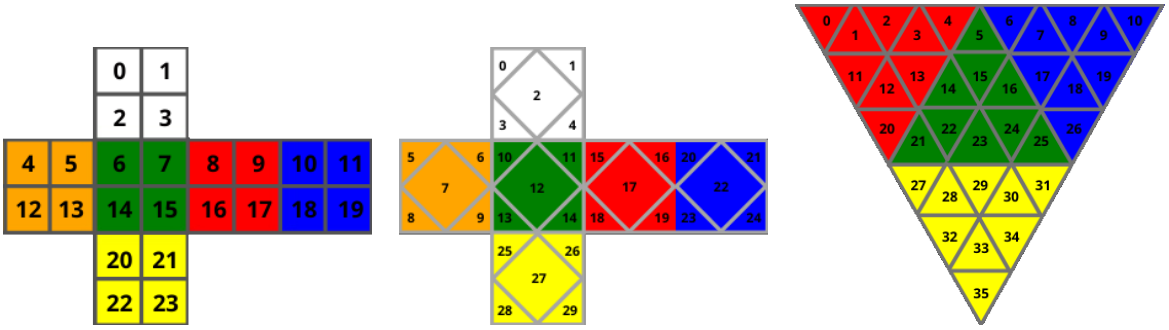
For the environment, and OpenAI Gym[21] package named RubiksCubeGym was built. It contains seven different environments, one for each of the puzzles and solution method combinations mentioned earlier. Each environment supports three rendering methods— ANSI(text), RGB Array and human. The human rendering method displays the puzzles in a 2D projection which is commonly used among cubers to show scrambles. It can be seen in Figure 4.

The environment package made use of Gym, NumPy and OpenCV as dependencies. Gym was used to integrate the environments with the OpenAI Gym API. NumPy was used to emulate the puzzles and their moves. It was also used to check the state of the puzzle and whether it met any goal or sub-goal. OpenCV was used for rendering images and RGB arrays. The scrambles for the puzzles were generated by an algorithm based on the WCA's open source scrambling software TNoodle [20]. Each puzzle is represented as Numpy *ndarray* and each state is given a unique number. This number and a text based representation of the puzzle are exposed through the *observation* and *info* parameters in the package's API. This API also exposes whether the attempt is complete (by either solving the puzzle or reaching the maximum number of steps(250) allowed) through the *done* parameter. The reward is decided based on the current state of the puzzle and exposed via the *reward*. As the different methods for solving the different puzzles have different steps, the reward for the sub-goals vary. However, the total positive reward for solving any puzzle is always 100. There is also a penalty of 1 per move that does not result in any goal or sub-goal being met. This forces the agent to search for efficient solutions. If any move results in a sub-goal being nullified, it is penalized for the same amount as the previously received reward to avoid the agent getting stuck in a loop of meeting the same sub-goal repeatedly[22].

B. Training

To speedup the operation of training the agent CS-RL was used. The algorithm split the total episodes equally per process and ran them parallely with a common Q-table. This can be seen in algorithm 1. The structure of the parallel section can be seen in Figure 5. The agent can create a new Q-table or use a previously saved table to train it further. It displays the 10,000 moving average for cumulative reward received per run and 10,000 moving success rate for reference. It also saves these along with the output Q-table for future use.

The training was carried out on a 4 core multi-threaded Haswell based i7, with eight processes running parallely but the algorithm will scale automatically according the improvements in the hardware given. To implement PQL, the memory sharing functions introduced in Python 3.8 were



(a) 2x2x2 Rubik's Cube (b) Skewb (c) Pyraminx

Fig. 4. Mapping of the various puzzles to their 2D projections

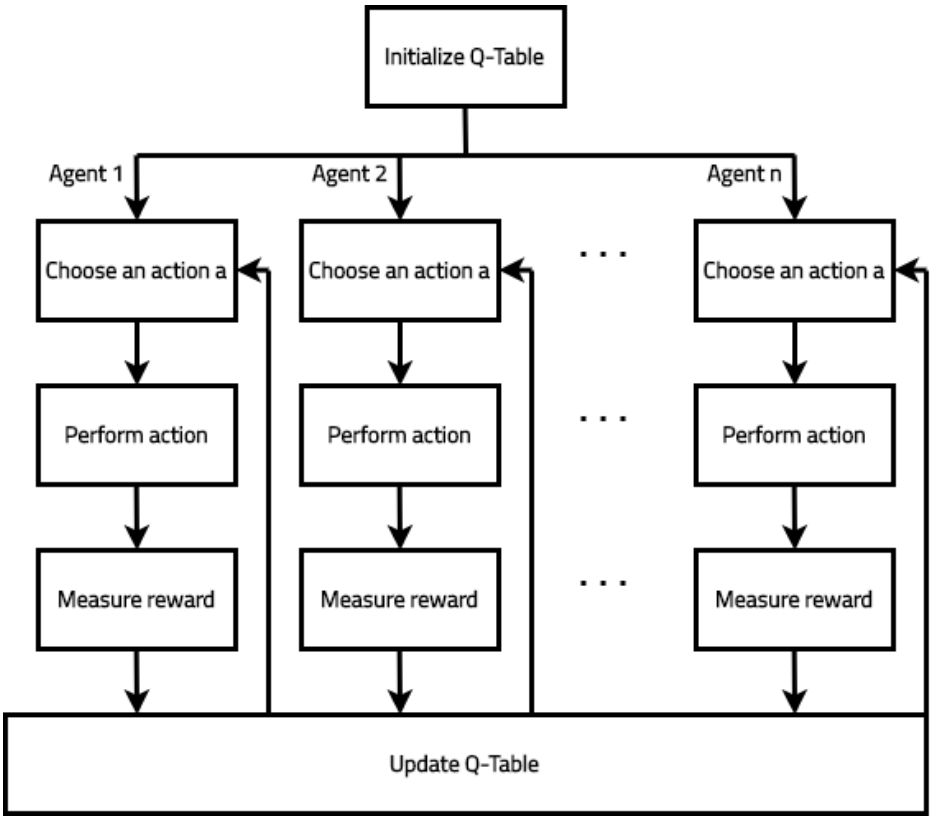


Fig. 5. Parallel Q-learning using Constant Share-Reinforcement Learning

used. This technique sped up the training significantly and given a processor with a higher core count, would allow even more complex puzzles to be solved.

C. Testing

To test the results, a validation set of known good scrambles was needed. This was created by downloading and extracting the WCA Competitive Results database[23]. A random subset of 10,000 samples was chosen for each puzzle and the resultant success rate, average reward and average number of moves were added to a tracker. The program could also render and solve scrambles using the generated Q-table as a means to visualize the agents work.

V. Results

The success rate, average reward and average move count for all the puzzles and methods can be seen in the following graphs and tables.

Algorithm 1 PQL using CS-RL

Require: n episodes, $n > 0$ and i agents, 1 per process
  Initialize Q(s,a) arbitrarily
  for each process **do**
    for n/i episodes **do**
      Initialize s
      while s is not terminal **do**
        Choose a from s using policy derived from Q (e.g., ϵ-greedy)
        Take action a, observe r, s'
        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$
      end while
    end for
  end for

TABLE I
RESULTS OF THE VARIOUS 2X2X2 POCKET RUBIK'S CUBE AGENTS

Training Size	Method	Success rate	Average reward	Average move count
2000000	lbl	49.89	-86.1917	136.5806
	none	66.9	-31.9635	99.5325
	ortega	18.48	-188.8264	207.4912
3000000	lbl	85.26	32.4786	53.6339
	none	99.8	79.5309	21.2671
	ortega	53.18	-54.1187	127.4008
5000000	lbl	98.12	77.5661	21.5351
	none	100.0	83.681	17.3190
	ortega	84.53	31.9869	53.3884
7500000	lbl	99.77	84.2557	16.512
	none	100.0	85.1919	15.8081
	ortega	99.09	81.5391	18.5418
10000000	lbl	100.0	85.5475	15.4525
	none	100.0	85.5908	15.4092
	ortega	99.95	85.2556	15.6939

TABLE II
RESULTS OF THE VARIOUS PYRAMINX AGENTS

Training Size	Method	Success rate	Average reward	Average number of moves
500000	lbl	8.54	-221.2995	229.9249
	none	32.68	-142.075	175.0818
600000	lbl	4.25	-235.6532	239.9457
	none	80.6	15.0487	66.3573
700000	lbl	28.78	-153.2174	182.2852
	none	97.52	73.6991	24.7961
800000	lbl	20.06	-182.2324	202.493
	none	99.62	82.0213	18.5949
900000	lbl	2.68	-240.8829	243.5898
	none	99.97	84.6026	16.3671
1000000	lbl	37.63	-122.8171	160.8234
	none	100.0	85.615	15.385
1500000	lbl	62.08	-39.7088	102.4096
	none	100.0	87.6315	13.3685
2500000	lbl	88.77	51.071999	38.5857
	none	100.0	89.3179	11.6821

TABLE III
RESULTS OF THE VARIOUS SKEWB AGENTS

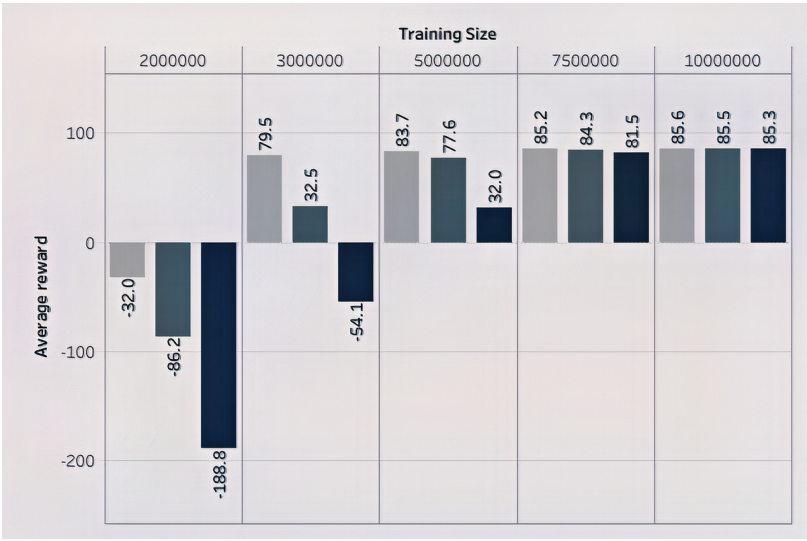
Training Size	Method	Success rate	Average reward	Average number of moves
2000000	none	54.91	-72.3876	127.8467
	sarah	8.25	-222.4332	230.7657
3000000	none	99.88	81.9239	18.9549
	sarah	37.21	-125.7149	163.297
5000000	none	100.0	86.4634	14.5366
	sarah	99.18	83.4465	16.7253
7500000	none	100.0	88.1116	12.8884
	sarah	99.89	87.7215	13.1674
10000000	none	100.0	88.7571	12.2429
	sarah	100.0	88.712	12.288

As we can see in Figure 6 and corresponding Table. I, the agent with no sub-goals learns the fastest. It is able to reach a success rate of 99.8% in just 3 million episodes. The training time needed was less than 6 hours for this result. It needed an average of 21 moves to solve the cube from any state. The agents with sub-goals based on the LBL and Ortega methods take much longer. They reach a similar success rate after 5 and 7.5 million episodes respectively. The agent based on the Ortega method requires only 18.5 moves on average to solve the puzzle compared to the 21 of the remainder when compared at a similar success rate.

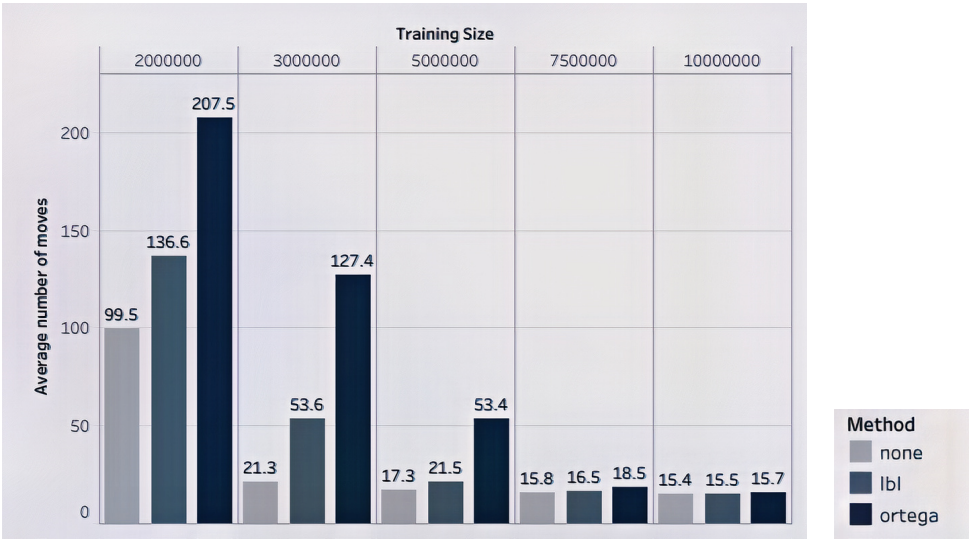
For the Pyraminx, the agent with no sub-goals reaches a success rate of 97.5% in just 700k episodes as seen in Figure 7 and corresponding Table. II. It requires an average of 25 moves to solve the puzzle. For the LBL method, the agent reaches a success rate of 88.77% in 2.5 million episodes. This is due to the fact that the agent would get stuck after solving the first layer and needed a lot more exploration time to reach the final solved state. The penalty for leaving the sub-goal was higher than the reward for solving the puzzle at certain stages after applying the discount factor γ. This can be seen in the dip in the success rate in the middle. This issue was overcome when the number of episodes n was large as the exploration time was also large.

The results for the Skewb were similar to the 2x2x2 Pocket Rubik's Cube. As seen in Figure 8 and corresponding Table. III, the agent without any sub-goals was the quickest to learn. It took only 3 million episodes to reach a success rate of 99.88% and needed an average of 19 moves. The agent with sub-goals based on Sarah's method (advanced) needed 5 million episodes to reach a similar success rate. However, it took an average of 16 moves to do so. To summarize the

(a) Success Rate



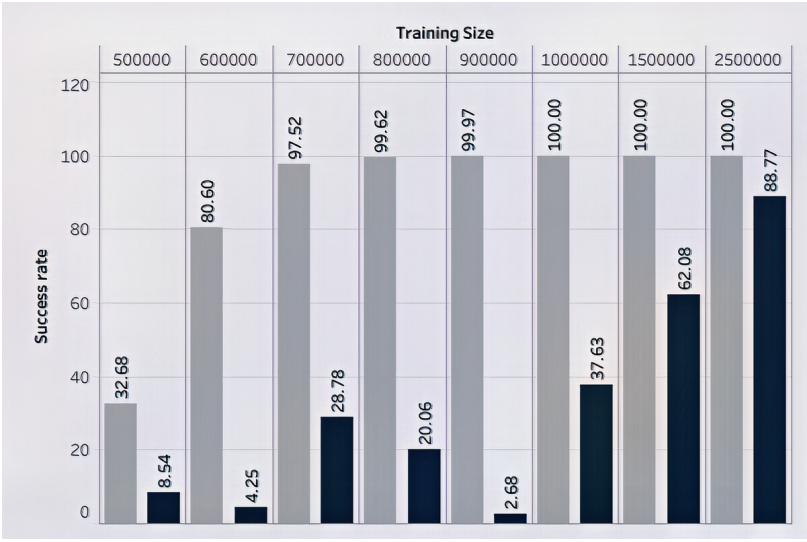
(b) Average Reward



(c) Average Move Count (d) Legend

Fig. 6. Results of the various 2x2x2 Pocket Rubik's Cube Agents

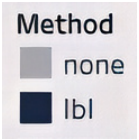
results, we were able to solve all the puzzles using PQL in a reasonable training time, given the hardware used. We also outperformed the NN based approach used in Karmakar 2020[12]. There is also a clear trend where the agents with



(a) Success Rate

(b) Average Reward

(c) Average Move Count



(d) Legend

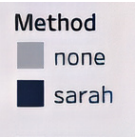
Fig. 7. Results of the various Pyraminx Agents

sub-goals performed worse than without agents. This can be attributed to the fact where the agents were penalized for each move they made and as meeting the sub-goals required a few extra moves, this slowed down their learning. While

(a) Success Rate

(b) Average Reward

(c) Average Move Count



(d) Legend

Fig. 8. Results of the various Skewb Agents

sub-goals help humans to keep track of the state of the puzzle and learn how to solve it quicker, this was not the case for the PQL agents. These agents were only given a limited action set for the 2x2x2 and Skewb to remove redundancy in the

observation set. While this reduced complexity of actions, it resulted in more steps being required. In the case of the Skewb, the reduction was needed in order to follow the WCA notation which only has four actions instead of the eight action. However, in the case of the 2x2x2, it was done purely to reduce the complexity of Q-table for performance gains in training time.

The validation of PQL based solutions is compared with DeepCube [11] for 2x2x2 Pocket Rubik's Cube puzzle. Authors have implemented DeepCube approach using Autodidactic Iteration which is supervised learning procedure to train a deep neural network. In our experiment, randomly scrambled 50 puzzles with increasing depths have provided for both the approaches. The result shows higher success ratio of solving puzzles in PQL than DeepCube. The comparison graph is shown in Figure 9.

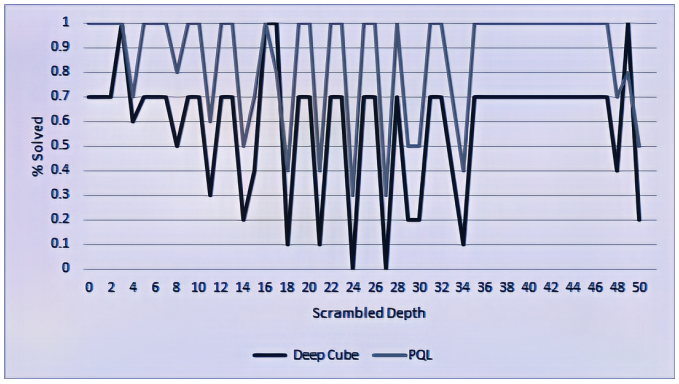


Fig. 9. Success Rate of Solved 2x2x2 Pocket Rubik's Cube

VI. Conclusions and Future Work

The aim of this paper was to emulate and solve Twisty Puzzles, with and without sub-goals, using PQL and to measure its impact. Using CS-RL, the agents learned to solve three puzzles— the 2x2x2 Pocket Rubik's Cube, the Skewb and the Pyraminx with a 100% success rate in a few hours on mediocre hardware. This is much better than the previous DRL approach [12] that was only able to achieve a 75% success rate when the scramble consisted of more than four moves. We also found that the use of sub-goals was detrimental to the agent, possibly due to factors specific to Twisty Puzzles and our technique of penalizing unnecessary moves as mentioned in Section V which might not apply to other MDPs. The PQL approach is compared with Deep Learning based network and obtained higher success rate.

There is a large scope in the exploration of CS-RL to solve more complex problems as it outperformed its DRL counterpart significantly. A study to measure its scalability for much larger problems such as the 3x3x3 Rubik's Cube could be performed. Different approaches to train the agents could also be studied such as to incentivise the use of commutators and conjugates to allow the agent to be more general and transferable across puzzles. There is also scope for future work to explore the impact of sub-goals for other kinds of combinatorial optimization problems. Our approach can also be used to solve 4x4x4 cube and other puzzles. The approach can be improved by finding suitable optimizations in modifying the sub-goals.

References

[1] Y. Hiroshima, "On reinforcement learning methods for generating train marshaling plan considering group layout of freight cars," *IAENG International Journal of Computer Science*, vol. 39, no. 3, pp. 239–245, 2012.

[2] H. Xuan, X. Zhao, J. Fan, Y. Xue, F. Zhu, and Y. Li, "Vnf service chain deployment algorithm in 5g communication based on reinforcement learning." *IAENG International Journal of Computer Science*, vol. 48, no. 1, pp. 1–7, 2021.

[3] Y. Yamaguchi, N. Shigei, and H. Miyajima, "Air conditioning control system learning sensory scale based on reinforcement learning," in *Proceedings of the International MultiConference of Engineers and Computer Scientists*, vol. 1, 2015.

[4] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016.

[5] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse *et al.*, "Dota 2 with large scale deep reinforcement learning," *arXiv preprint arXiv:1912.06680*, 2019.

[6] Elizabeth, "Self-taught ai is best yet at strategy game go," *Nature*, 2017.

[7] *Dota 2 Player Count*. [Online]. Available: https://steamdb.info/app/570/graphs/

[8] *OpenAI Five*. [Online]. Available: https://openai.com/blog/openai-five/

[9] O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. M. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell *et al.*, "Alphastar: Mastering the real-time strategy game starcraft ii," *Deep Mind blog*, p. 2, 2019.

[10] A. P. Badia, B. Piot, S. Kapturowski, P. Sprechmann, A. Vitvitskyi, D. Guo, and C. Blundell, "Agent57: Outperforming the atari human benchmark," 2020.

[11] S. McAleer, F. Agostinelli, A. Shmakov, and P. Baldi, "Solving the rubik's cube without human knowledge," *arXiv preprint arXiv:1805.07470*, 2018.

[12] D. Karmakar, S. Naskar, M. Burada, S. Kaithwar, and V. Singh, "Solution to 2x2x2 rubik's cube using autodiadactic iteration: A deep reinforcement algorithm," 2020.

[13] T. Tateyama, S. Kawata, and T. Shimomura, "Parallel reinforcement learning systems using exploration agents and dyna-q algorithm," in *SICE Annual Conference 2007*. IEEE, 2007, pp. 2774–2778.

[14] M. Kushida, K. Takahashi, H. Ueda, and T. Miyahara, "A comparative study of parallel reinforcement learning methods with a pc cluster system," in *2006 IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, 2006, pp. 416–419.

[15] A. M. Printista, M. L. Errecalde, and C. I. Montoya, "A parallel implementation of q-learning based on communication with cache," *vol. 1, no. 6*, 2002.

[16] M. Camelo, J. Famaey, and S. Latré, "A scalable parallel q-learning algorithm for resource constrained decentralized computing environments," in *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, 2016, pp. 27–35.

[17] Takeshi Tateyama, Seiichi Kawata, and Toshiki Shimomura, "Parallel reinforcement learning systems using exploration agents and dyna-q algorithm," in *SICE Annual Conference 2007*, 2007, pp. 2774–2778.

[18] J. Scherphuis. [Online]. Available: https://www.jaapsch.net/puzzles/

[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.

[20] *WCA Regulations and Guidelines*. [Online]. Available: https://www.worldcubeassociation.org/regulations

[21] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, vol. 1, no. 1, 2016.

[22] A. D. Laud, "Theory and application of reward shaping in reinforcement learning," Tech. Rep., 2004.

[23] *WCA Results Export*. [Online]. Available: https://www.worldcubeassociation.org/results/misc /export.html