A Deep Reinforcement Learning Approach for Solving Pyraminx Without Human Knowledge

Ritesh Goru 160070048

Devang Sailor 173079010

Sudarshan P 17307r003

Avinash Reddy 183070010

1 Problem Statement

Pyraminx is a 3 dimensional combinatorial puzzle with 4 triangular sides. There are 16 possible movements which include 8 trivial actions. There are less than 10^6 possible combinations, compared to 3.47x10^6 in 2x2x2 rubik's cube or 43.25x10^{18} combinations in 3x3x3 rubik's cube. Each puzzle in rubik's puzzle family is denoted with a god's number which describes the minimum number of steps required to reach the terminal state, for 3x3x3 rubik's it is 21 steps and the same for pyraminx is 11 steps. Our aim is to train a DRL agent to solve pyraminx for upto 'N' perturbations, with an attempt to maximize 'N' considering the limits of computation complexity.

2 Our Approach

2.1 Modelling the Pyraminx

We have utilized pyraminx simulator framework from (Ludisposed, 2018), which models all the actions. For the project we would consider using only 8 of the actions, ignoring the other 8 trivial actions which links tip pieces with the center pieces. This reduces further the possible combinations by a factor of 3^4. The solved state of pyraminx is showed in Figure 1.

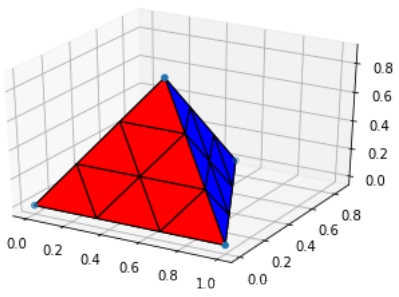


Figure 1: Pyraminx visualization tool

2.2 Algorithm

Our task can be divided into 2 sub-tasks: Prediction and Solver. In prediction, the aim is to find the value function of the given state. For this task we have attempted three ways to predict values, which includes tabular method, neural network model trained on True values and autodidactic iteration (ADI). Tabular method is done by computing exact values by traversing through all possible combinations of pyraminx model using a tree search. We also use these exact true values to train neural network, which again was used as a value approximation. We use autodidactic iteration (ADI), a supervised learning algorithm described by McAleer et al. which trains a joint value and policy network (McAleer et al., 2018). The ADI algorithm and neural network models used are explained in section 5. In Solver, the goal is to achieve the objective of solving the cube with any possible path using the values and policy from the trained neural network. For Solver, we have used greedy and two variants of MCTS-solver (Mark H.M.Winands, 2008) (Silver et al., 2018). As a baseline for our solver, we will be using the greedy breadth-first search for evaluation.

2.3 Novelty

ADI augumented with MCTS had been implemented on 3x3x3 Rubik's cube (McAleer et al., 2018) and 2x2x2 Rubik's cube (Nicholas W. Bowman, 2018).We try to implement on the Pyraminx using ADI augmented with MCTS (Silver et al., 2018) and ADI augmented with MCTS solver (Mark H.M.Winands, 2008). We also attempt on training models on ADI which were pre-trained with true value.

3 Evaluation

For each model and solver combination, we provide a scrambled pyraminx puzzle of depth d, where each puzzle is generated from d consecutive actions on the solved one. We have taken the depth value d varying from 1 to 8. We have computed percentage solved cases i.e. total number of solved scrambles in 100 random scrambles for each d depth within k number of steps, where k is defined as $k = 6d+1$.

We also compare the performance of the algorithm in terms of number of steps taken by the model to solve the puzzle across all methods for a given scrambled case.

4 Modelling as an MDP

The states of the MDP is the matrix of size $9 \times 4 \times 4$. 9 blocks on each face, 4 faces and 4 colours. $|\mathcal{S}| = 933,120$. It has a total of 8 actions excluding the 4 corner single tetrahedron rotation. The set of actions are $\mathcal{A} = [u, b, l, r, u', b', l', r']$. $u-$ upper 2 level tetrahedron, $b-$ backside 2 level tetrahedron, $l-$ left 2 level tetrahedron and $r-$ right 2 level tetrahedron. $*$ action is clockwise and $*'$ is anti-clockwise rotation. Reward is 1 for the solved state [Fig 1] and -1 for every other state.

5 Methods

5.1 Predictor

5.1.1 Neural Network Architechture

Two different linear feedforward neural networks, shown in Fig 2a and 2b, were used to predict both the value and the policy for a given state. As a single network was used for both value and policy, we have used Leaky-Relu and Sigmoid activation functions for value and policy respectively. A sum of mean square loss and cross entropy was used to back-propagate and train our networks. Adam optimizer was used for all the results we implemented. The weights of the neural network were initialized using Glorot uniform initialization.

v, p are the value of given input state and probability vector of given input state.

5.1.2 Neural Network Models

The Models we used are described as follows have are:

1. **Model 1:** NN 2 trained using ADI ($N = 4$)
2. **Model 2:** We had calculated tabular true value data of all states.
3. **Model 3:** NN 1 trained on true values tabular data.
4. **Model 4:** NN 2 trained on true values tabular data.
5. **Model 5:** Retraining Model No. 3 on ADI ($N = 8$).

5.1.3 ADI

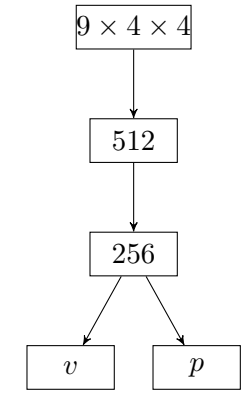
A neural network is trained by algorithm called Auto Didactic Iteration(ADI). From a solved state, a total of l samples were collected iteratively shown in Fig 3. Following same procedure for k times, we get $N = l \times k$ samples for training DNN.

In each iteration of ADI, training samples are generated starting from the solved puzzle. This will ensure that some of the training inputs will be close enough for some positive rewards at the early stage of training. Targets are created by performing a depth-1 breadth-first search (BFS) from each training sample. The current value network is used to estimate each child's value. The value target for each sample is the maximum value and reward of each of its children, and the policy target is the action which led to this maximal value.

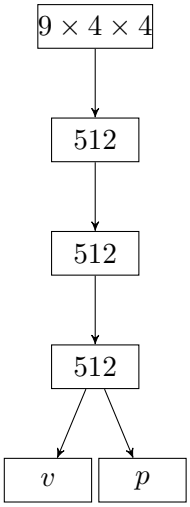
Algorithm 1 Auto Didactic Iteration

Initialization: θ initialized using Glorot initialization
repeat
  for $x_i \in X$ **do**
    for $a \in A$ **do**
      $(v_{x_i}(a), p_{x_i}(a)) \leftarrow f(A(x_i, a))$
    end
    $y_{vi} \leftarrow max_a(R(A(x_i, a)) + v_{x_i}(a))$
    $y_{pi} \leftarrow argmax_a(R(A(x_i, a)) + v_{x_i}(a))$
    $Y_i \leftarrow (y_{v_i}, y_{p_i})$
  end
  $\theta' \leftarrow train(f_\theta, X, Y)$
  $\theta \leftarrow \theta'$
until *iterations* = *M*;

A state sampling technique is followed in this algorithm to generate samples which ensures convergence. The samples are generated by Depth-1



(a) Neural Network(NN) 1



(b) Neural Network(NN) 2

Figure 2: Neural Network Models

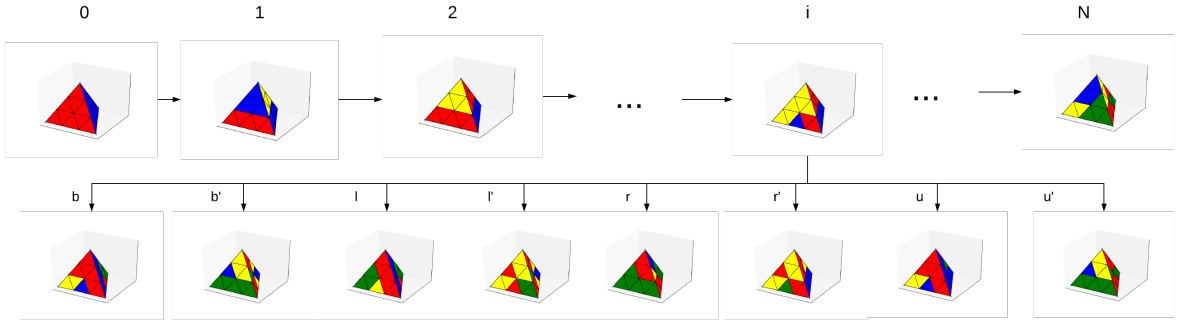


Figure 3: Collecting Samples

Breadth First search. For training we used Adam optimizer with a mean squared error loss for the value and softmax cross entropy loss for the policy. To ensure convergence, we assigned a higher training weight to samples that are closer to the solved cube compared to samples that are further away from solution. We assign a loss weight to each sample $i, W_{x_i} = \frac{1}{D_{x_i}}$

5.2 Solver

We have used greedy solver. Given a state to the DNN f_θ, outputs the probability values of all actions and value of that state. Moving in the direction of the highest probable action, we try to solve the Pyraminx. We use this greedy solver as baseline to compare our future solvers. We used a variant of MCTS solver which was first proposed by (Silver et al., 2018) for AlphaZero. This variant doesn't follow UCT version of MCTS solver rather it follows a PUCT approach(Silver et al., 2018). We build a search tree T starting from the state s_o. Every node in Tree T is associated with four parameters $N_{s_t}(a), L_{s_t}(a), P_{s_t}(a), W_{s_t}(a)$ No. of visits, Virtual Loss, Prior Probabilities, Total value for each action a respectively . The search strategy is building a tree by choosing action $A_t = argmax_a U_{s_t}(a) + Q_{s_t}(a).U_{s_t} = cP_{s_t}(a)\sqrt{\frac{\sum_{a'} N_{s_t}(a')}{1+N_{s_t}(a)}}, Q_{s_t} = W_{s_t}(a) - L_{s_t}(a)$. Once the leaf node s_τ is reached, the state is expanded by all children $s' = \{A(s_\tau, a), \forall a \in A\}$. Then for each child s', $N_{s_t}(a) = 0, L_{s_t}(a) = 0, P_{s_t}(a) = p_{s'}, W_{s_t}(a) = 0$. Calculate the value and Probability of s_τ, $(v_{s_\tau}, p_{s_\tau}) = f_\theta(s_\tau)$. Backup all the values on all visited states: $W_{s_t}(A_t) \leftarrow max(W_{s_t}(A_t), v_{s_\tau}), N_{s_t}(A_t) \leftarrow N_{s_t}(A_t) + 1, L_{s_t}(A_t) \leftarrow L_{s_t}(A_t) - \nu$.. Here, the tree looks like the a node with 8 branches without

any sub branches. Maximum value along the tree is chosen to update the values. we move along the action corresponding to branch with highest Q value. This MCTS solver approach is derived from (McAleer et al., 2018) The depth of Tree T, τ is selected based on computational strength. The psuedo code for MCTS variant is shown in Algorithm based on 2

Algorithm 2 MCTS variant 1

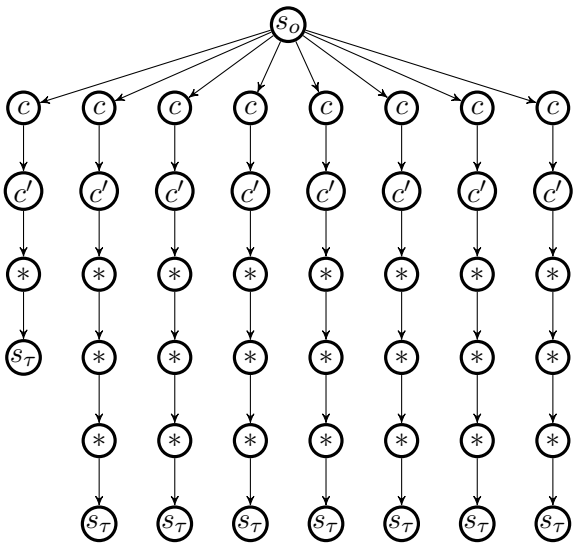
Input: State s_o
Output : $argmax_c Q_{\forall c \in s_o}$
for $a \in A$ **do**
  Child c of $s_o \leftarrow$A(s$_o$, a).
  if *child is Terminal state* **then**
    return index(a)
  else
    continue
  end
end
for *child c of* s_o **do**
  repeat
    $A_t = argmax_a U_{s_t}(a) + Q_{s_t}(a)$.
    child c' of $c \leftarrow$A(c,a)
    if *c' is Terminal state* **then**
      Continue
    else
      $c \leftarrow c'$
    end
  until *depth* = d;
end
for *leaf node* s_τ *of Tree* T **do**
  for $a \in A$ **do**
    $s_t = A(s_\tau, a)$
    $N_{s_t}(a) = 0, L_{s_t}(a) = 0, P_{s_t}(a) = p_{s'}, W_{s_t}(a) = 0$
  end
end
for *nodes in Tree* T **do**
  $W_{s_t}(A_t) \leftarrow max(W_{s_t}(A_t), v_{s_\tau})$
  $N_{s_t}(A_t) \leftarrow N_{s_t}(A_t) + 1$
  $L_{s_t}(A_t) \leftarrow L_{s_t}(A_t) - \nu$..
end
In $(\circ)_{s_t}(A_t)$ represents parameter $(\circ)$ of state action pair (s_t, A_t), t is MCTS depth index.
where, $U_{s_t} = cP_{s_t}(a)\sqrt{\frac{\sum_{a'} N_{s_t}(a')}{1+N_{s_t}(a)}}$,
$Q_{s_t} = W_{s_t}(a) - L_{s_t}(a)$

Algorithm 3 MCTS variant 2

Input: State s
for $a \in A$ **do**
  Child c of $s \leftarrow$A(s,a).
  if *child is Terminal state* **then**
    return index(a)
  else
    continue
  end
end
if *s in visited states* **then**
  for *child c of s* **do**
    for $a \in A$ **do**
      $s' =$ child of $c = (A(c, a))$
      $N_{s'}(a) = 0, L_{s'}(a) = 0, P_{s'}(a) = p_c, W_{s'}(a) = 0$
    end
  end
  for *all visited states* **do**
    $W_{s_t} \leftarrow max(W_{s_t}, v_{A(s_o,A_t)})$
    $N_{s_t} \leftarrow N_{s_t} + 1$
    $L_{s_t} \leftarrow L_{s_t} - \nu$..
  end
else
  return $A_t = argmax_a U_s(a) + Q_s(a)$.
end
s_t is all the visited state at step t the agent took to solve pyraminx. ν is hyperparameter to control virtual loss.

We can have many types of MCTS variant solvers by varying the backup strategy.

5.2.1 An example of MCTS Variant 1 tree



6 Results

We had tried Algorithm 2,3 MCTS variants and Greedy solvers on all 5 Predictor models showed in Figure 4. We present the graphs of percentage of solved cubes against the scarambling distance upto 8 (scrambling distance is the number of random actions applied on the solved pyraminx to scramble). For each depth we have plotted the number of solves from 100 scrambles as described in evaluation. The reason for taking upto 8 scrambling depth is because the God's Number of Pyraminx is 11. God's Number is the minimum number of steps taken to achieve the solved state from any given state.

7 Observations

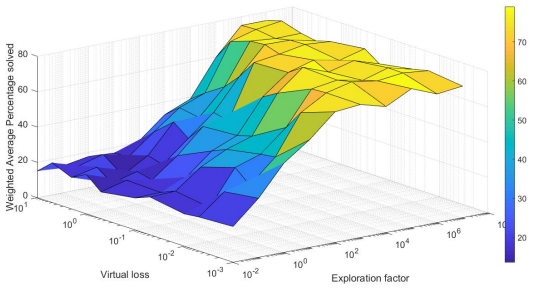


Figure 5: Weighted Average Percentage Solved plot for Tabular (Predictor) and MCTS variant 2 (Solver) case

The tabular true value function model(Model 2) with greedy solver gives the best result. Given any state of pyraminx, it solves in minimum number of moves. MCTS variants due to exploration they are taking more number of moves to solve.

For MCTS algorithm there are 2 more hyper parameters, first is exploration factor and other is virtual loss. The exploration factor tells how much to trust the Predictor output probability vector and virtual loss is to put the action value within the limits. To verify this statement, we have used Tabular as our predictor, as it will give the exact prediction. From the 3D plot shown in figure 5, that with the increase in the exploration factor the percentage solved metric is increasing largely compared to the virtual loss. We have used these plots to choose these hyper parameters for MCTS solver.

8 Challenges

The models were hard to train neural networks for following reasons:

1. The values for given MDP are really close for consecutive states so it is hard to train models to predict exact value of the states, and with this approximation of value function, MCTS doesn't give exact results.

2. The model has $9 \times 4 \times 4$ input size which is one-hot encoding of each state, which gives 2^{144} possible combinations but due to puzzle constraints there are only approx. 10^6 possible combinations. This resulted in difficulty in training models without any form of encoding of state variable.

9 Group Contributions

All code used for this project can be found on github at: https://github.com/BlackWingedKing/CS747_Project. All members of the project have contributed equally.

10 Credits

This document has been adapted from the instructions for ACL 2019 proceedings.

References

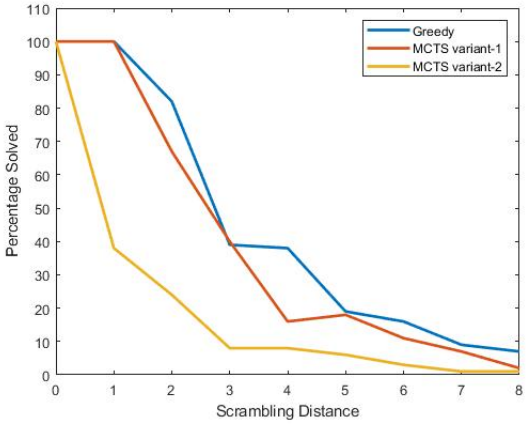
Ludisposed. 2018. *Answer on stack exchange, CodeFights: Pyraminx puzzle*.

Jahn-Takeshi Saito Mark H.M.Winands. 2008. *Monte-Carlo Tree Search Solver*.

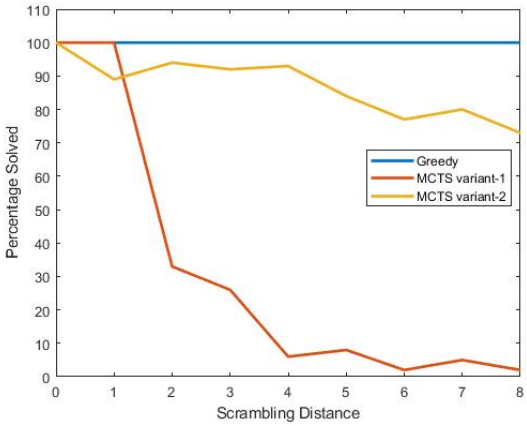
Stephen McAleer, Forest Agostinelli, Alexander Shmakov, and Pierre Baldi. 2018. Solving the Rubik's Cube Without Human Knowledge. *arXiv e-prints*, page arXiv:1805.07470.

Robert M.J. Jones Nicholas W. Bowman, Jessica L. Guo. 2018. *BetaCube: A Deep Reinforcement Learning Approach to Solving 2x2x2 Rubik's Cubes Without Human Knowledge*.

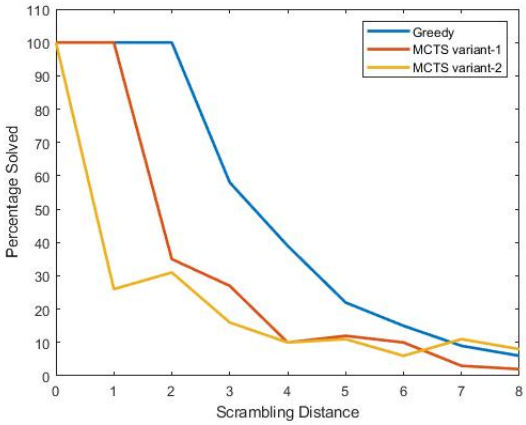
David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2018. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144.



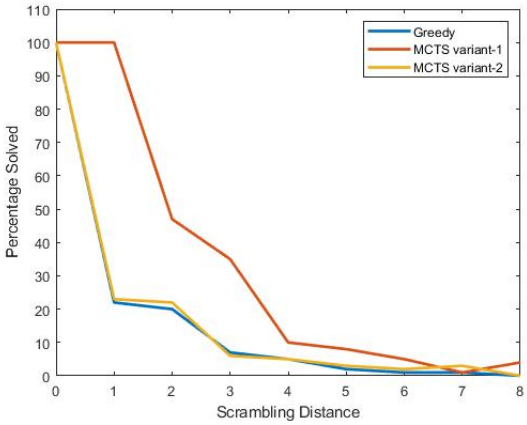
(a) Model 1: NN2 trained with ADI (depth-4)



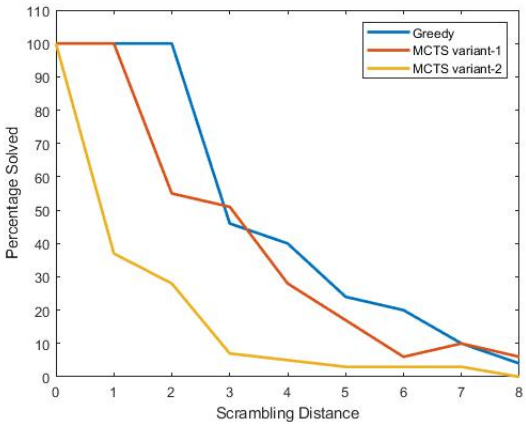
(b) Model 2: Tabular



(c) Model 3: NN1 trained on Tabular data



(d) Model 4: NN2 trained on Tabular data



(e) Model 5: Pre-trained NN1 trained with ADI (depth-8)

Figure 4: Results