SOLVING THE RUBIK'S CUBE WITH APPROXIMATE POLICY ITERATION

Stephen McAleer, Forest Agostinelli, Alexander Shmakov, Pierre Baldi

THE PROBLEM

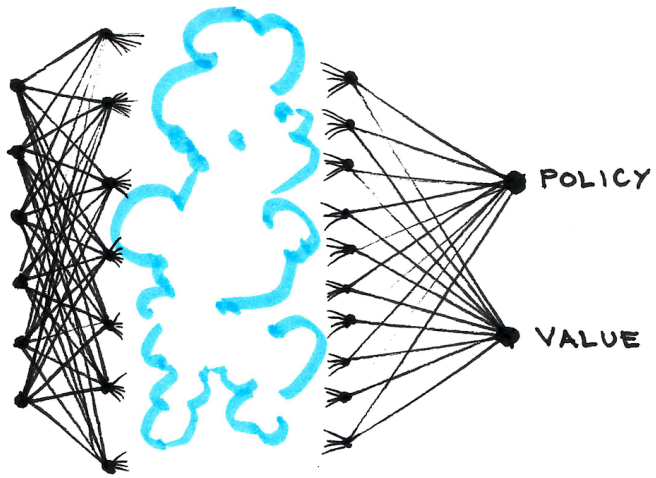
- Astronomically large state space (4.2 x 10^{12} different states for 3x3x3 rubik cube)
- Single player game
- Only single state is considered solved
- A sequence of random moves is unlikely to end in the solved state (sparse reward environment)
- Only recently derived methods can solve the cube in minumum amount of moves from any starting configuration
- As the length of the edges and number of dimensions increase, complexity increases exponentially
- No clear way to apply current DPI algorithms such as AlphaZero or ExIt to sparse-reward environments such as the Rubik's Cube.
- Approximating the value function is very difficult because even a naive Monte-Carlo approach would not work since it will never encounter the solved state.

RELATED WORK

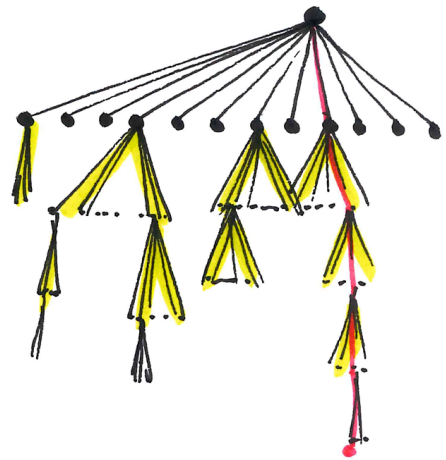
- In 2014, it was shown that any valid cube can be optimally solved with at most 26 moves in the quarter-turn metric, or 20 moves in the half-turn metric
- Algorithms used by machines to solve the Rubik's Cube rely on hand-engineered features and group theory to systematically find solutions (Kociemba)
- In 1985 it was shown that iterative deepening A* could be used to solve the Rubik's Cube (combined with pattern databases) (Korf)
- Supervised learning with hand-engineered features (Brunetto & Trunda, 2017)
- Attempts have been made to solve the Rubik's Cube through evolutionary algorithms (Smith et al., 2016; Lichodzijewski & Heywood, 2011)

- Approximate Policy Iteration [Bertsekas & Tsitsiklis, 1995]
- ExIt (Expert Iteration) ["Thinking Fast and Slow with Trees and Deep Learning", Anthony et al. 2017]

METHODS



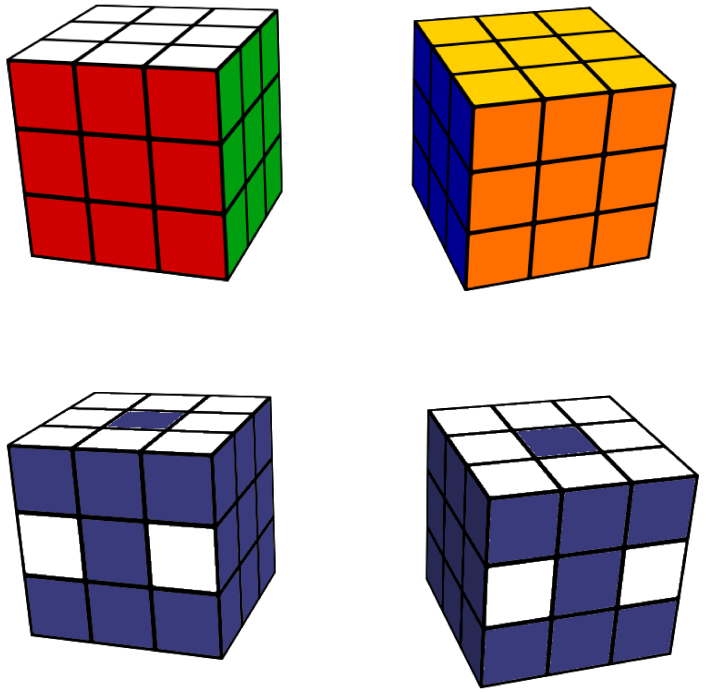
(FAST POLICY)
APPRENTICE



(SLOW POLICY)
EXPERT

THE CUBE

- The Rubik's Cube consists of 26 smaller cubes called cubelets.
- There are 54 stickers in total
- Each sticker is uniquely identifiable based on the type of cubelet the sticker is on and the other sticker(s) on the cubelet.
- The dimensionality of the representation can be reduced by focusing on the position of only one sticker per cubelet
- Ignore the redundant center cubelets and only store the 24 possible locations for the edge and corner cubelets.
- This results in a 20x24 state representation
- Moves are represented using face notation: $\mathrm{F}, \mathrm{B}, \mathrm{L}, \mathrm{R}, \mathrm{U}, \mathrm{D}$
- A clockwise rotation is represented with a single letter, whereas a letter followed by an apostrophe represents a counter-clockwise rotation
- At each timestep, t, the agent observes a state $s_t \in S$ and takes an action $a_t \in A$ with $A := \{\mathrm{F}, \mathrm{F}', \ldots, \mathrm{D}, \mathrm{D}'\}$
- After selecting an action, the agent observes a new $s_{t+1} = A(s_t, a_t)$ and receives a scalar reward, $R(s_{t+1})$, which is 1 if s_{t+1} is the goal state and −1 otherwise.



METHODS

Approximate Policy Iteration (Brief Overview)

bles. Instead, we consider the **Approximate Policy Iteration** algorithm (Bertsekas & Tsitsiklis, 1996) defined iteratively by the two steps:

- *Approximate policy evaluation*: for a given policy π_k, generate an approximation V_k of the value function V^{π_k}
- *Policy improvement*: generate a new policy π_{k+1} greedy with respect to V_k:

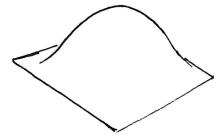
$$\pi_{k+1}(i) = \arg\max_{a \in A} \sum_{j \in X} [r(i,a,j) + \gamma\, p(i,a,j) V_k(j)]$$

These steps are repeated until no more improvement of the policies is noticed (using some evaluation criterion). Empirically, the value functions V^{π_k} rapidly improve in the first iterations of this algorithm, then oscillations occur with no more performance increase.

Taken from "Error Bounds for Approximate Policy Iteration", Remi Munos

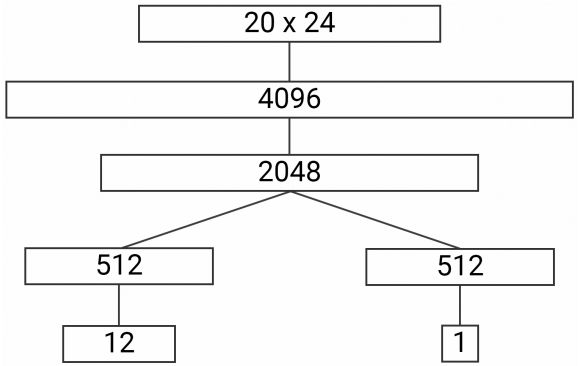
METHODS
Autodidactic Iteration

- The sampling distribution is generated by starting from the solved state and randomly taking actions
- Cubes closer to the starting state are weighted more heavily – a loss weight 1 / $D(x_i)$ is assigned to each sample, where $D(x_i)$ is the number of scrambles it tool to generate it

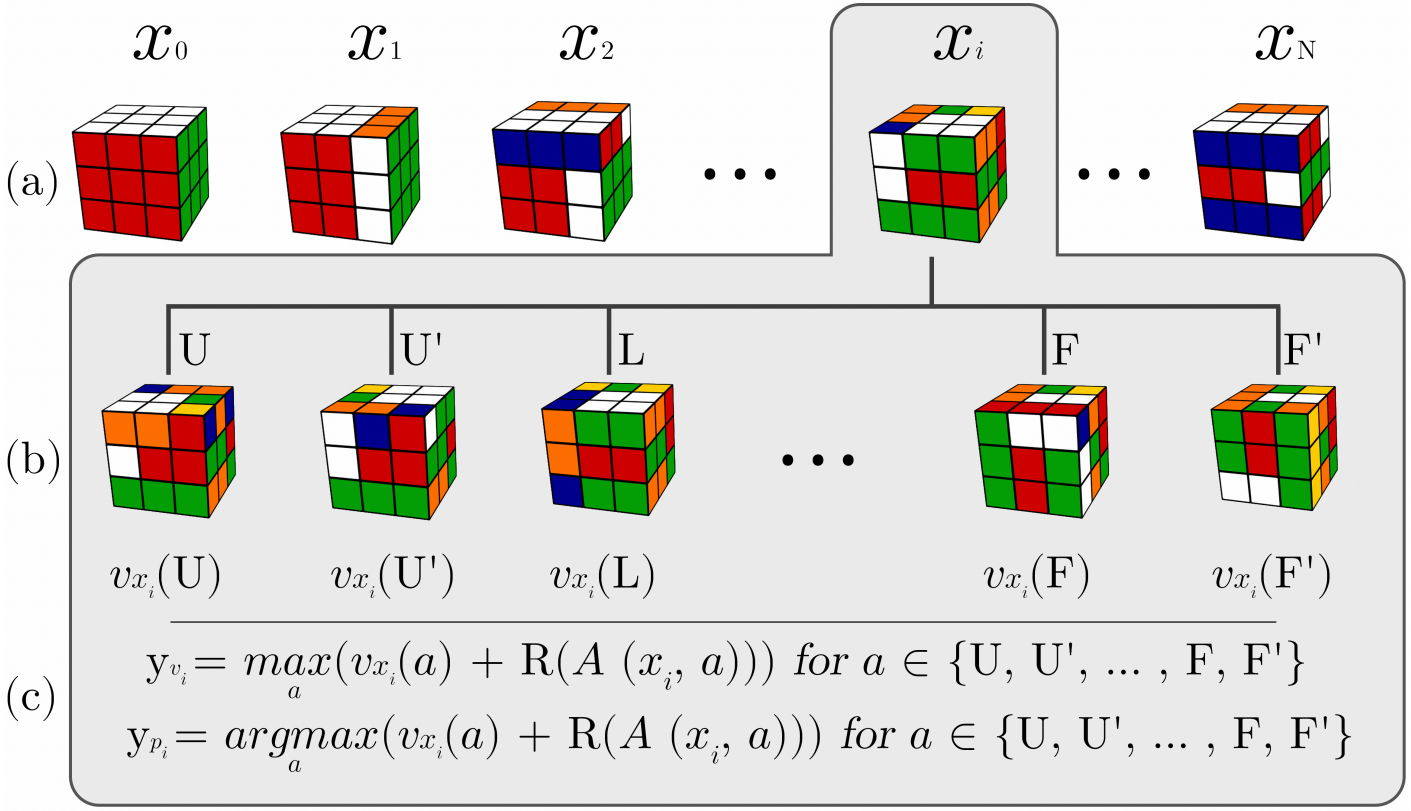


Algorithm 1: Autodidactic Iteration

Initialization: θ initialized using Glorot initialization
repeat
  $X \leftarrow$ N scrambled cubes
  for $x_i \in X$ **do**
    for $a \in \mathcal{A}$ **do**
      $(v_{x_i}(a), \boldsymbol{p}_{x_i}(a)) \leftarrow f_\theta(A(x_i, a))$
    $y_{v_i} \leftarrow \max_a(R(A(x_i, a)) + v_{x_i}(a))$
    $\boldsymbol{y}_{p_i} \leftarrow \text{argmax}_a(R(A(x_i, a)) + v_{x_i}(a))$
    $Y_i \leftarrow (y_{v_i}, \boldsymbol{y}_{p_i})$
  $\theta' \leftarrow train(f_\theta, X, Y)$
  $\theta \leftarrow \theta'$
until $iterations = M$;



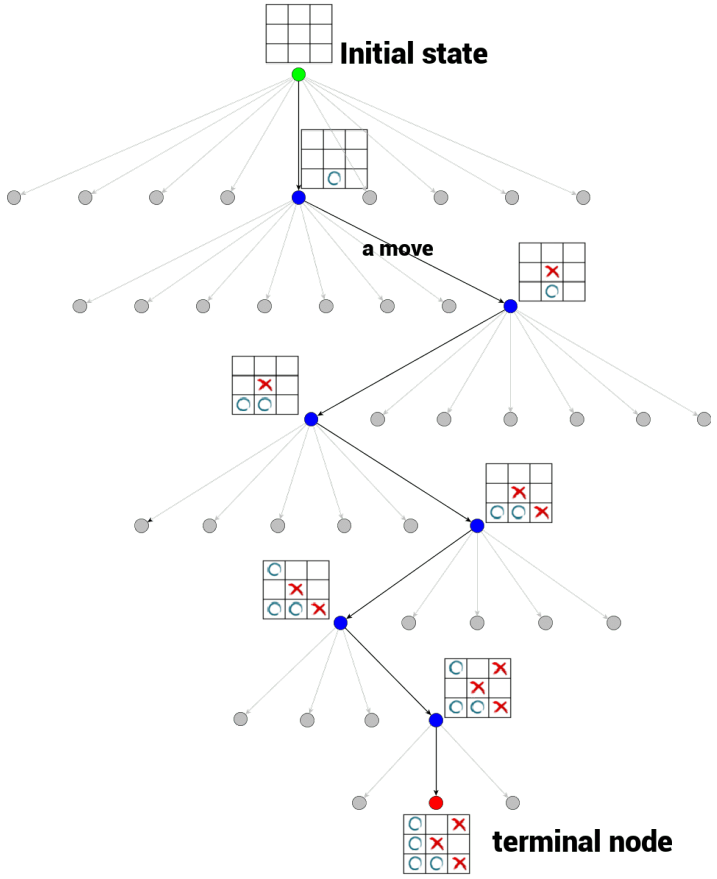
METHODS
Autodidactic Iteration



METHODS

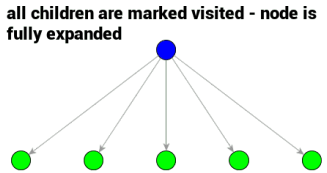
Monte-Carlo Tree Search (Brief overview)

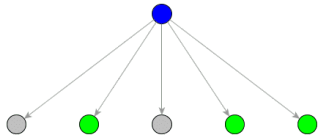
- Every node represents a state in the game
- Root node represents the initial state
- PLAYOUT is a sequence of moves that start from the current node and end in games's terminal state
- A node is VISITED if a PLAYOUT was started at least once in it
- A node is FULLY EXPANDED if all it's children are VISITED
- A node is TERMINAL if it has no children and the game cannot continue from it
- A node is a LEAF if a PLAYOUT has just started for it
- Simulation always starts at the node that has not been visited previously
- During simulation the moves are chosen with respect to a ROLLOUT POLICY FUNCTION
- Nodes chosen by ROLLOUT POLICY FUNCTION during simulation are not considered VISITED
- BACKPROPAGATION is a traversal back from the leaf node (where simulation started) up to the root
- Ultimate goal is to find the most promising next move
- Simulates the game many times

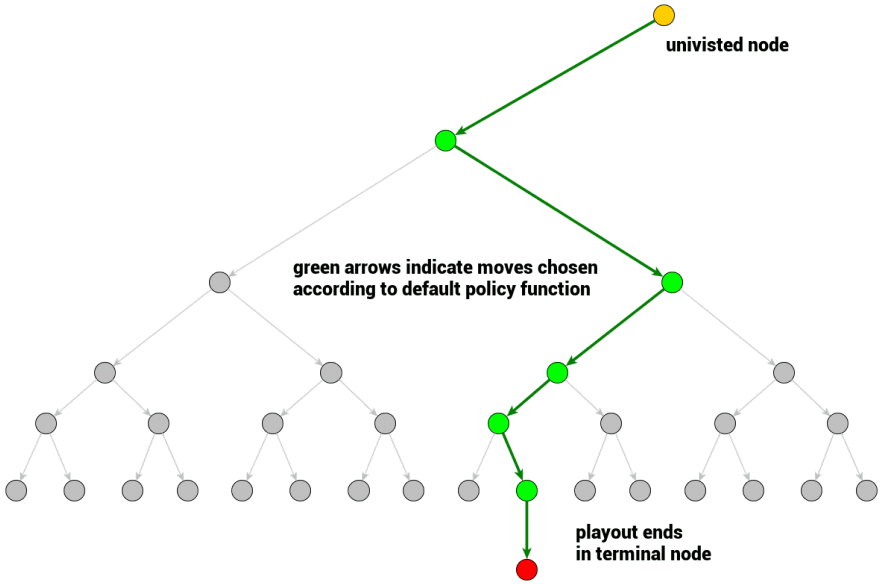


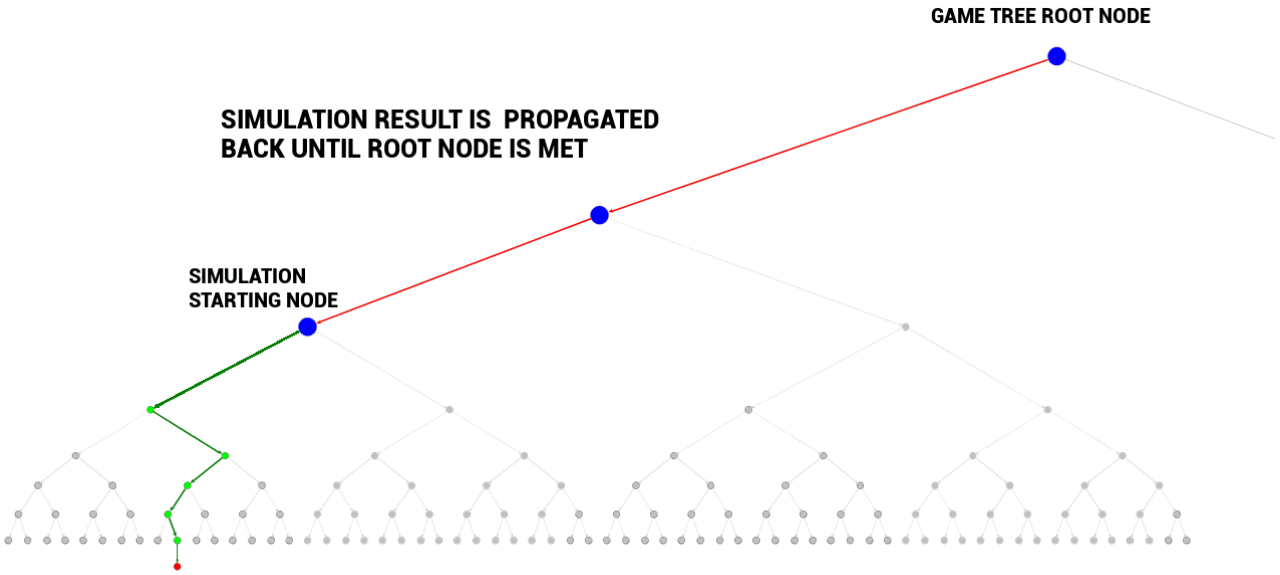
METHODS

Monte-Carlo Tree Search (Brief overview)









METHODS

Monte-Carlo Tree Search (Brief overview)

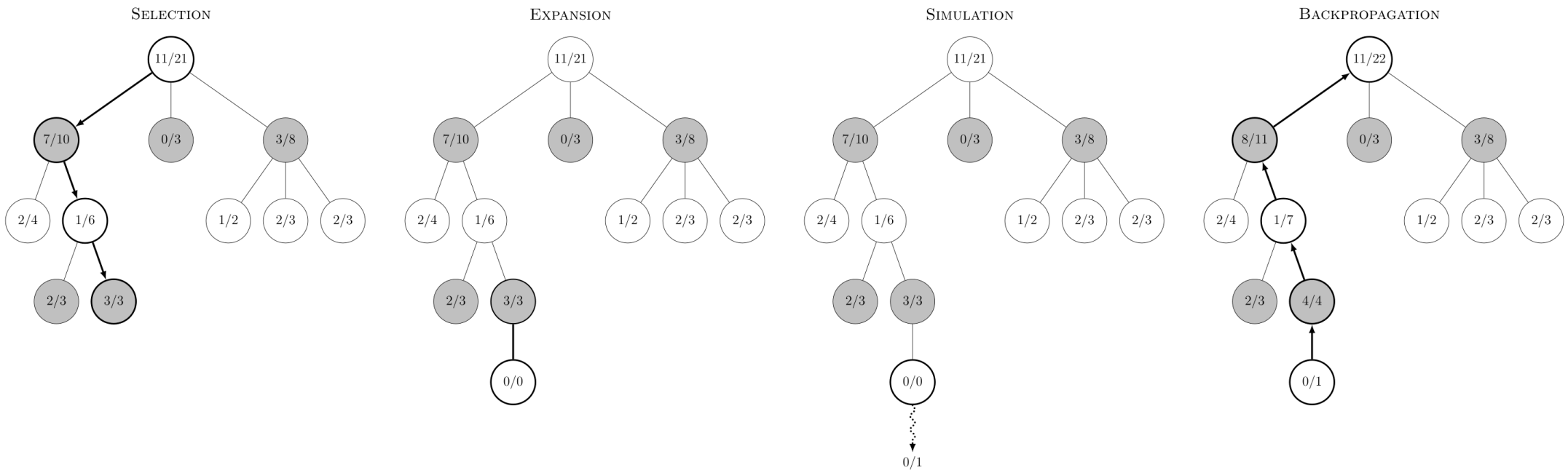
- Node statistics
  - Total number of visits $\mathrm{N}(\mathrm{s})$
  - Total simulation reward $\mathrm{Q}(\mathrm{s})$
- BACKPROPAGATION updates $\mathrm{N}(\mathrm{s})$ and $\mathrm{Q}(\mathrm{s})$
- Exploitation follows nodes with high $\mathrm{Q}(\mathrm{s})$, exploration nodes with low $\mathrm{N}(\mathrm{s})$
- Upper Confidence Bound applied to trees (UCT) is a function that let us choose the next node among visited nodes to traverse through

$$UCT(s, s_i) = \frac{Q(s_i)}{N(s_i)} + c\sqrt{\frac{\log(N(s))}{N(s_i)}}$$

- Node maximizing UCT is the one to follow during Monte Carlo Tree Search tree traversal
- In competitive games $\mathrm{Q}(\mathrm{s_i})$ is always computed relative to player who moves at node i

METHODS

Monte-Carlo Tree Search (Brief overview)



METHODS

Solver (Asynchronous Monte Carlo Tree Search augmented with trained neural network $\mathbf{f}_\theta$)

1. A search tree is build iteratively beginning with starting state $\mathbf{T} = \{ s_0 \}$
2. Simulated traversals are performed until reaching a leaf node of $\mathbf{T}$
3. Each state has a memory attatched to it:
    1. $N_s(a)$ – number of times action ‘a’ is taken from state ’s’
    2. $W_s(a)$ – the maximum value of action ‘a’ from state ‘s’
    3. $L_s(a)$ – virtual loss for action ‘a’ from state ‘s’
    4. $P_s(a)$ – the prior probability of action ‘a’ from state ‘s’
4. Every simulation starts from the root node and iteratively selects actions by following a tree policy until an unexpanded leaf node, s_T , is reached.
5. Once a leaf node, s_τ , is reached, the state is expanded by adding the children of s_τ , $\{A(s_\tau, a), \forall a \in A\}$, to the tree $\mathbf{T}$
6. Next, the value and policy are computed: $(v_{s\tau}, p_{s\tau}) = \mathbf{f}_\theta (s_\tau)$ and the value is backed up on all visited states in the simulated path
7. The simulation is performed until either s_τ is the solved state or the simulation exceeds a fixed maximum computation time.
8. If s_τ is the solved state, the tree is expanded, adding all children of any unexplored states
9. Breadth-first search is performed to find the shortest path from starting state to the solution

METHODS
Solver details

- (Step 4) The tree policy proceeds as follows:

  For each timestep t, an action is selected by choosing $A_t = \text{argmax}_a \; U_{st}(a) + Q_{st}(a)$ where

$$U_{st}(a) = c\, P_{st}(a) \frac{\sqrt{\sum_{a'} N_{st}(a')}}{(1+N_{st}(a))}$$

$$Q_{st}(a) = W_{st}(a) - L_{st}(a)$$

$$L_{st}(A_t) \leftarrow L_{st}(A_t) + \nu$$

- (Step 5) Children updates:

  For $0 \leq t \leq \tau$, the memories are updated:

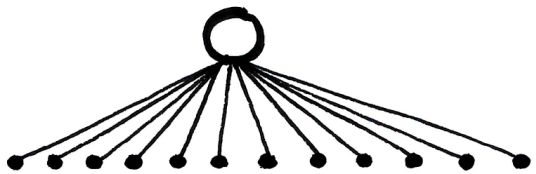
$$W_{st}(A_t) \leftarrow \max\left(W_{st}(A_t), v_{s\tau}\right)$$

$$N_{st}(A_t) \leftarrow N_{st}(A_t) + 1$$

$$L_{st}(A_t) \leftarrow L_{st}(A_t) - \nu$$

- Unlike other implementations of MCTS, only the maximal value encountered along the tree is stored, not the total value. This is because the Rubik's Cube is deterministic and not adversarial

WALK THROUGH

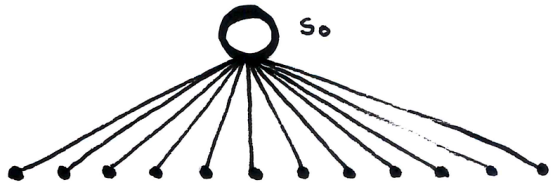


LEAF NODE REACHED
ADD CHILDREN
INITIALIZE

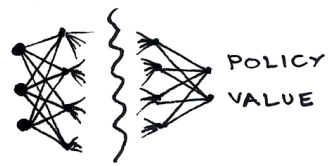
$W_{s'}(\cdot) = 0$

$N_{s'}(\cdot) = 0$

$L_{s'}(\cdot) = 0$



COMPUTE THE VALUE AND POLICY FOR s_0 USING $f_\theta(s_0)$



UPDATE (INITIALIZE) THE PRIOR PROBABILITIES FOR EACH CHILD OF s_0 USING π_{s_0}

BACKPROPAGATE s_0's VALUE (EMPTY STEP)

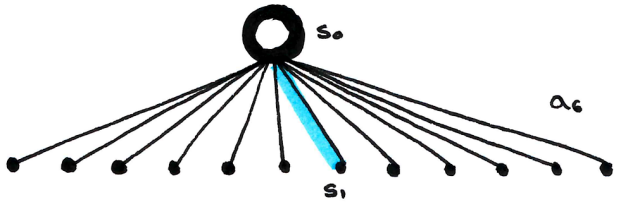
UPDATE

$W_{s_0}(a) \leftarrow \max(W_{s_0}(a), v)$

$N_{s_0}(a) \leftarrow N_{s_0}(a) + 1$

$L_{s_0}(a) \leftarrow L_{s_0}(a) - v$

WALK THROUGH



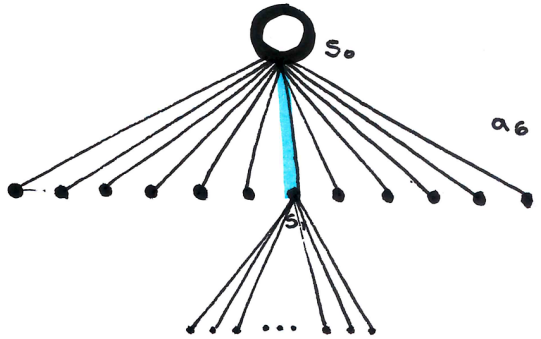
- START AT ROOT (s_0)
- CHOOSE NEXT NODE USING

$$\text{argmax}_a \; U_s(a) + Q_s(a)$$

- LEAF NODE REACHED (s_1)

$$U_{s_0}(a) = c P_{s_0}(a) \frac{\sqrt{\sum_{a'} N_{s_0}(a')}}{1 + N_{s_0}(a)}$$

$$Q_{s_0}(a) = W_{s_0}(a) - L_{s_0}(a)$$



- EXPAND (ADD CHILDREN)
- INITIALIZE $W_{s'} = 0$, $N_{s'} = 0$, $L_{s'} = 0$
- COMPUTE VALUE AND POLICY FOR s_1 USING $f_\theta(s_1)$
- INITIALIZE THE PRIOR PROBABILITY FOR EACH CHILD OF s_1
- BACKPROPAGATE s_1's VALUE

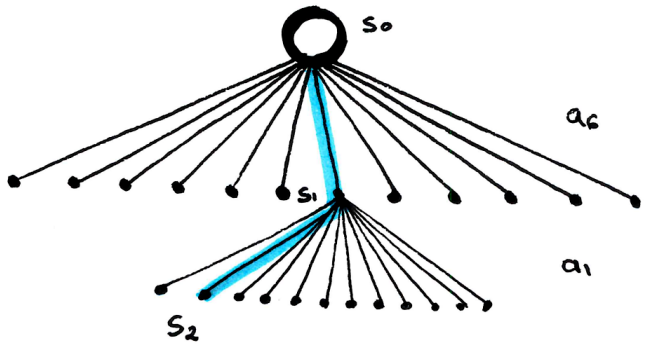
$$W_{s_0}(a_6) = \max\left(W_{s_0}(a_6), v\right)$$

- UPDATE

$$N_{s_0}(a_6) = 1$$

$$L_{s_0}(a_6) = -v$$

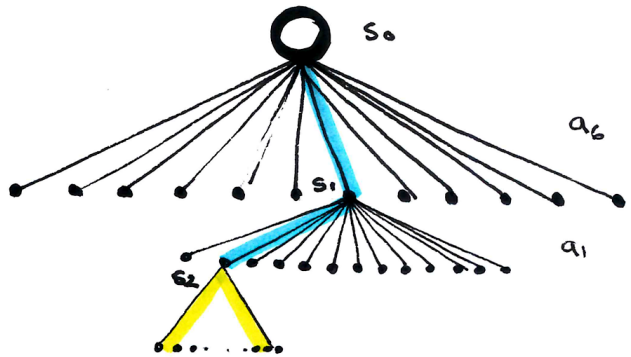
WALK THROUGH



- TRAVERSE A PATH $s_0 \rightarrow s_1 \rightarrow s_2$ ENDING IN LEAF NODE USING

$$\text{argmax}_a \; U_s(a) + Q_s(a)$$

AT EACH LEVEL OF THE TREE



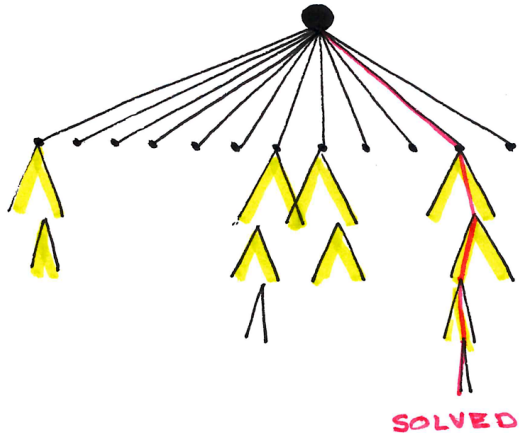
- EXPAND (ADD CHILDREN)
- INITIALIZE $W_{s'} = 0$, $N_{s'} = 0$, $L_{s'} = 0$
- COMPUTE VALUE AND POLICY FOR s_2 USING $f_\theta(s_2)$
- INITIALIZE THE PRIOR PROBABILITY FOR EACH CHILD OF s_2
- BACKPROPAGATE s_2's VALUE

$$W_{s_1}(a_1) = \max\left(W_{s_1}(a_1), v\right)$$
$$W_{s_0}(a_6) = \max\left(W_{s_0}(a_6), v\right)$$

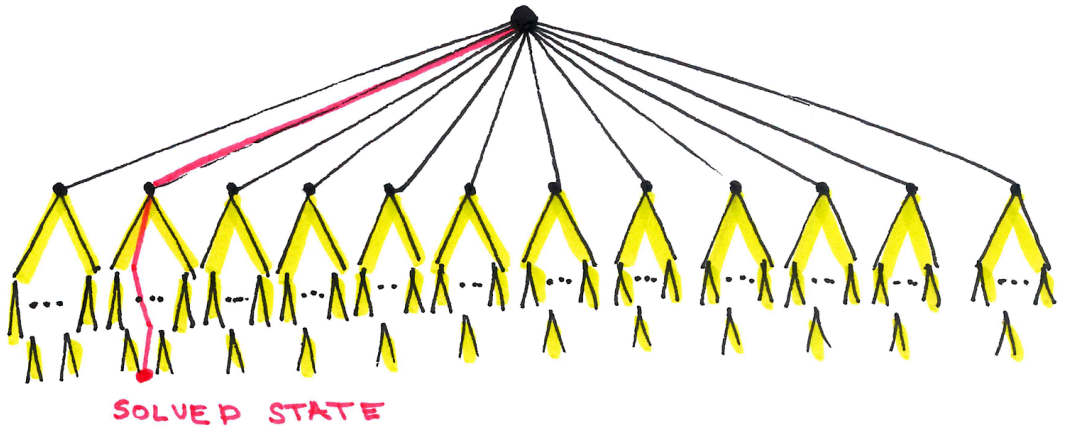
- UPDATE

$$N_{s_0}(a_6) = 2 \qquad L_{s_0}(a_6) = -2v$$
$$N_{s_1}(a_1) = 1 \qquad L_{s_1}(a_1) = -v$$

WALK THROUGH

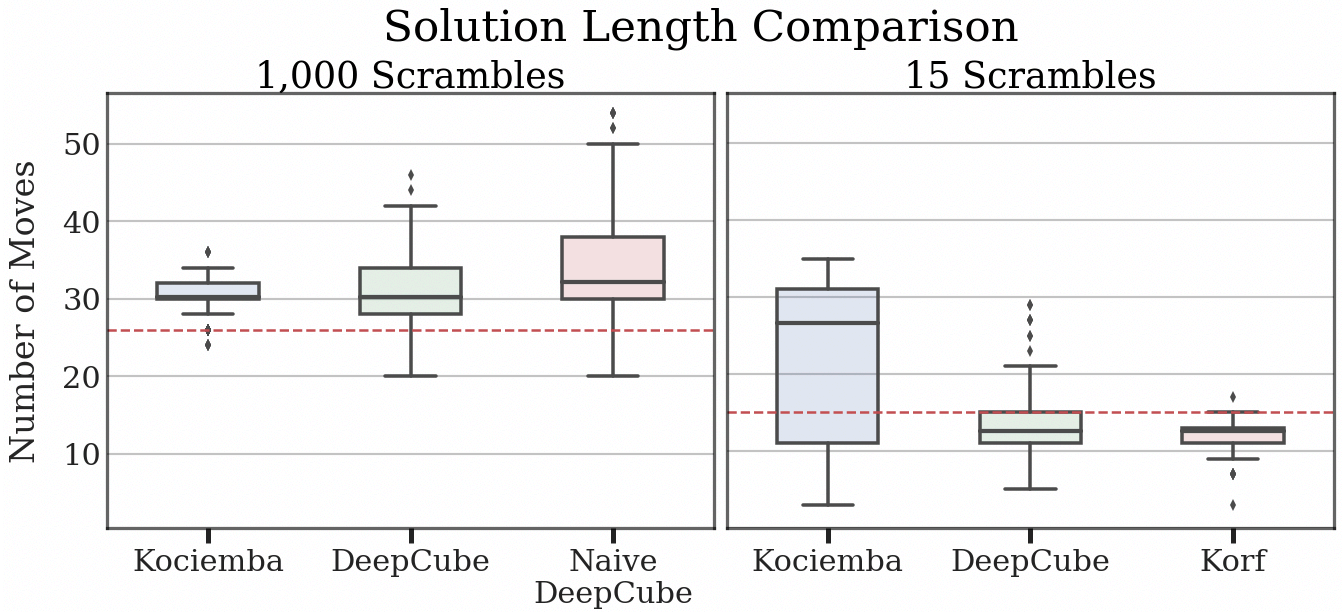


SUPPOSE THAT AFTER N SIMULATIONS WE FIND A (THE) SOLVED STATE 5 STEPS AWAY (MOVES) FROM STARTING CONFIGURATION (ROOT)



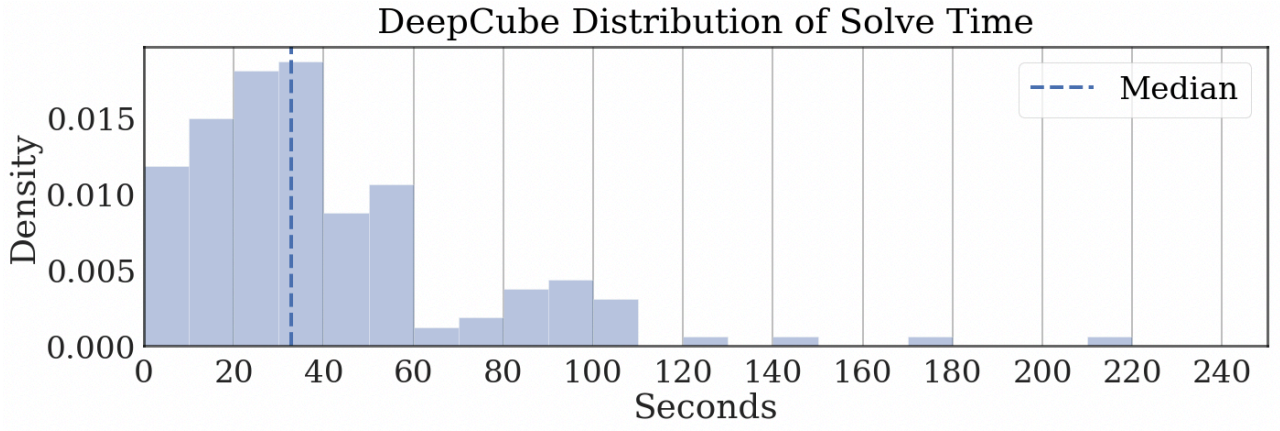
- DO BREADTH-FIRST SEARCH, EXPANDING ALL CHILDREN
- WE FIND THE SOLVED STATE 4 MOVES AWAY FROM STARTING CONFIGURATION
- ALGORITHM TERMINATES

RESULTS



- Kociemba solved each cube in under a second, while DeepCube had a median solve time of 10 minutes.
- Although DeepCube has a much higher variance in solution length, it was able to match or beat Kociemba in 55% of cases.
- BFS removes any trivial cycles in the path such as a move followed by its inverse, but it also improves solution lengths by finding a slightly more efficient path between states that are within 3 moves of each other.

RESULTS



Solver	Nodes	Seconds	Nodes/Sec	Mean Solution Length	Memory Requirement
Rokicki	1.86E+06	**2.74**	**1.86E+06**	**20.6**	182 GB
Korf	122 billion	40	2E+06	**20.6**	2 GB
DeepCube	**7.823E+03**	40	2.1E+03	30.6	1 GB
Kociemba	N/A	0.035	N/A	30.5	30 MB

- The median solve length for both DeepCube and Korf is 13 moves, and DeepCube matches the optimal path found by Korf in 74% of cases.
- The Korf optimal solver requires an average expansion of 122 billion different nodes for fully scrambled cubes
- DeepCube expands an average of only 7,823 nodes with a maximum of 24,175 expanded nodes on the longest running sample.

DISCUSSION

- DeepCube is based on similar principles as AlphaZero and ExIt
- One is not guaranteed to find a terminal state in the Rubik's Cube environment and therefore may only encounter rewards of -1, which does not provide enough information to solve the problem
- DeepCube addresses this by selecting a state distribution for the training set that allows the reward to be propagated from the terminal state to other states.
- The depth-1 BFS used to improve the policy can also be viewed as doing on-policy temporal difference learning with function approximation.
- DeepCube is able to teach itself how to reason in order to solve a complex environment with only one positive reward state using pure reinforcement learning.

THANK YOU !