Solving the Rubik's Cube Without Human Knowledge

Authors: Stephen McAleer, Forest Agostinelli, Alexander Shmakov, Pierre Baldi
Presenter: Stelios Andrew Stavroulakis

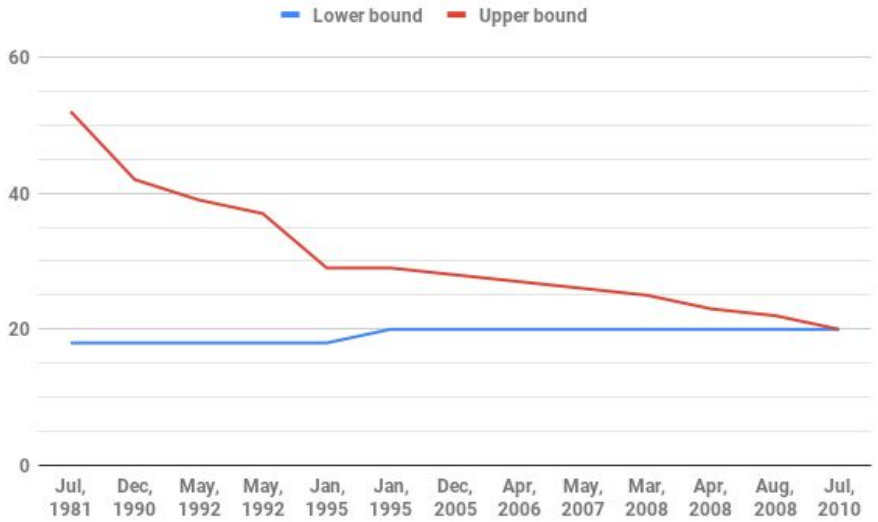
Why tackle the Rubik's Cube?

- Application of RL in combinatorial optimization
- Famous problems (same discipline):
  - Traveling Salesman Problem
  - Protein Folding Simulation
- If solve state available => training a value (+policy) function using ADI
  - For example, protein folding, the goal is to find the protein conformation with minimal free energy. We don't know what the optimal conformation is beforehand, but we can train a value network using ADI on proteins where we know what their optimal conformation is.
- Implementation in
  - Planning problems where environment has many states
  - Find goal but unaware what the goal is

Previous Methods

- Group Theory Utilization
  - **Kociemba's two-stage solver**
    - Maneuver the cube to a smaller group
    - Solve the (trivial) cube
    - No guarantee for optimal solution
  - **Korf's Algorithm (variation of A* heuristic)**
    - Identify a number of subproblems that are small enough to be solved optimally.
      - The cube restricted to only the corners, not looking at the edges
      - The cube restricted to only 6 edges, not looking at the corners nor at the other edges.
      - The cube restricted to the other 6 edges.
    - Although this algorithm will always find optimal solutions, there is no worst case analysis.
  - **DNN / Evolutionary Algorithms**
    - Usually fail to find a solution to randomly scrambled cubes

God’s Number


Lower bound
Upper bound
60
40
20
0
Jul, 1981
Dec, 1990
May, 1992
May, 1992
Jan, 1995
Jan, 1995
Dec, 2005
Apr, 2006
May, 2007
Mar, 2008
Apr, 2008
Aug, 2008
Jul, 2010

Why is this paper important

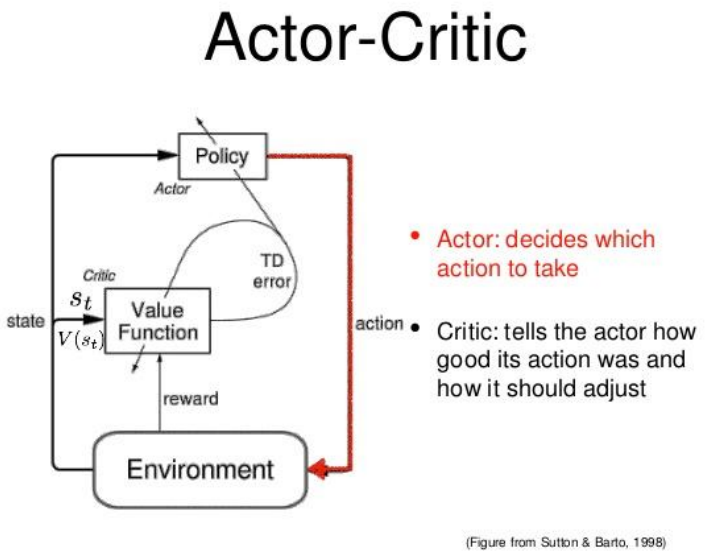
- Minimal Human Supervision
- No Domain Knowledge
- Sparse Reward Problem
- No Termination Guarantee

“Deep Reinforcement Learning relies heavily on the condition that an informatory reward can be obtained from an initially random policy”

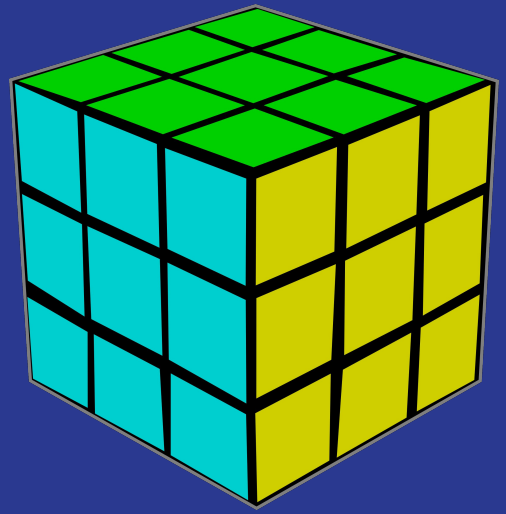
- Stephen McAleer

Rubik’s Cube

1. Combination Puzzle
2. Large State Space - 4.3 * 10^19
3. **Single Reward State** - solved state
4. Advantage actor-critic A3C would never solve - why?



Cube Representation



Action Space (12)

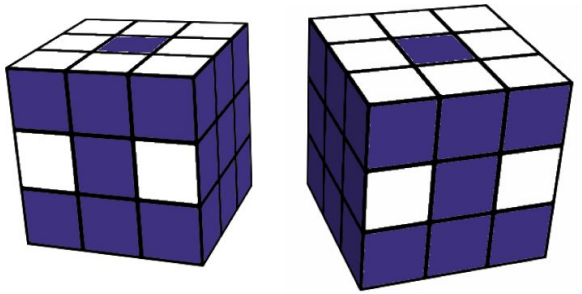
	Clockwise	Counterclockwise
Left	**L**	**L’**
Right	**R**	**R’**
Top	**T**	**T’**
Down	**D**	**D’**
Front	**F**	**F’**
Back	**B**	**B’**

State Space Representation Goals

- Avoid Redundancy
  - Color Recording: $10^{37} >> 10^{19}$
- Memory Efficiency
  - Save a large amount of different states in memory
- Performance of transformations
  - Compact representations require unpacking which hurts training speed
- Neural Network Friendliness
  - Tensors are friendly in encapsulating dimensional data

One-Hot Encoding (cubelets - positions)

	p_1	p_2	p_3	...	p_{24}
c_1	0	0	1	...	0
c_2	1	0	0	...	0
...	...	...	...	...	...
c_{20}	0	0	0	...	1



(c) Top view (d) Bottom view

Rewards

- Special state - $\mathbf{s_{solved}}$
- At each timestep:
  - $s_t \in S$
  - $a_t \in A$
- Translates to:
  - $s_{t+1} = A(s_t, a_t)$
- Reward:
  - $R(s_{t+1})$ = 1 if s is goal state
  - $R(s_{t+1})$ = -1 otherwise

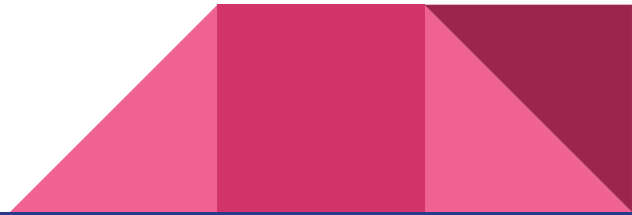
“From this single positive reward given at the solved state, DeepCube learns a value function. DeepCube improves its value estimate by first learning the value of states one move away from the solution and then building off of this knowledge to improve its value estimate for states that get progressively further away from the solution.”

Training Process

Autodidactic Iteration

Autodidactic Iteration

- Algorithm Inspired by policy iteration
  - The policy iteration algorithm manipulates the policy directly, rather than finding it indirectly via the optimal value function
  - Authors trained a joint value and policy network.
- ADI is an iterative supervised learning training process
- Algorithm starts from solved state and works backwards in order to create training examples



Algorithms Steps

1. Apply every possible transformations (12 in total) to the state s
2. Pass those 12 states to our current neural network, asking for value output.
   a. This gives us 12 values for every sub-state of s.
3. Target value for state s is calculated as $v_i = max_a(v(s_i,a)+R(A(s_i,a)))$,
   a. A(s, a) is the state after action a applied to the state s and
   b. R(s) equals 1 if s is the goal state and -1 otherwise.
4. Target policy for state s is calculated using the same formula, but instead of max we take argmax: $p_i = argmax_a\ (v(s_i,a)+R(A(s_i,a)))$.
   a. This just means that our target policy will have 1 at the position of maximum value for sub-state and 0 on all other positions.

Pseudocode:

Initialization: θ initialized using Glorot initialization
repeat
    X ← N scrambled cubes
    for ($x_i \in X$) **do** {
        for ($a \in A$) **do** {
            $(v_{xi}(a), p_{xi}(a)) \leftarrow f_\theta(A(x_i, a))$
            $y_{vi} \leftarrow \max_a(R(A(x_i, a)) + v_{xi}(a))$
            $y_{pi} \leftarrow \text{argmax}_a (R(A(x_i, a)) + v_{xi}(a))$
            $Y_i \leftarrow (y_{vi}, y_{pi})$
        }
    }
    $\theta' \leftarrow \text{train}(f_\theta, \mathbf{X}, Y)$
    $\theta \leftarrow \theta'$
until iterations = M;

ADI Figure

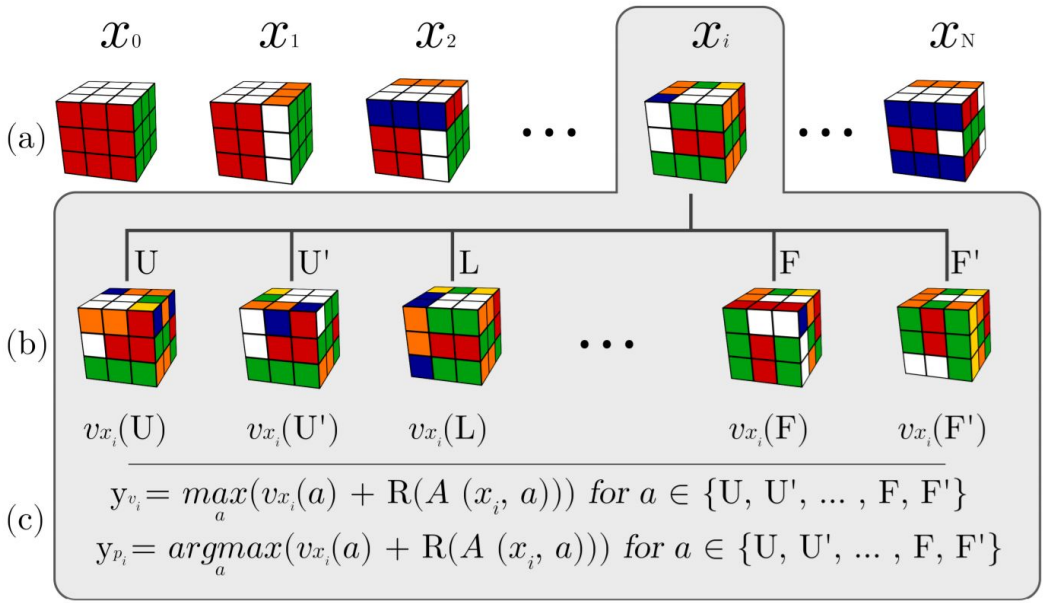
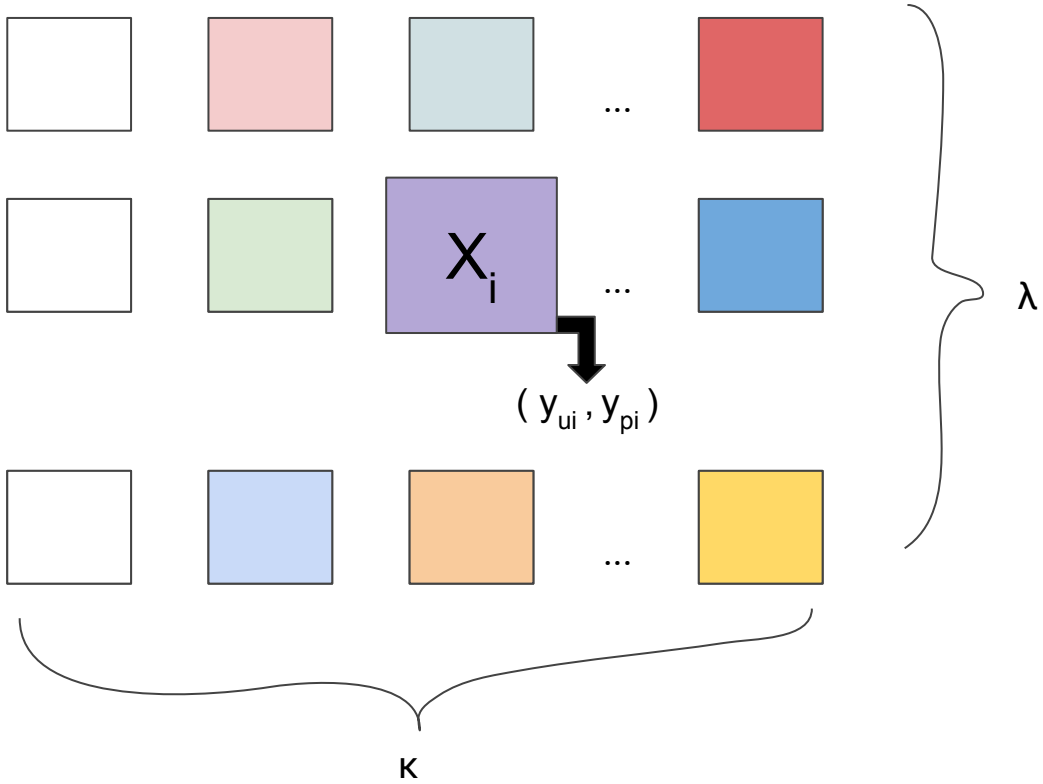


Figure 3: Visualization of training set generation in ADI. (a) Generate a sequence of training inputs starting from the solved state. (b) For each training input, generate its children and evaluate the value network on all of them. (c) Set the value and policy targets based on the maximal estimated value of the children.

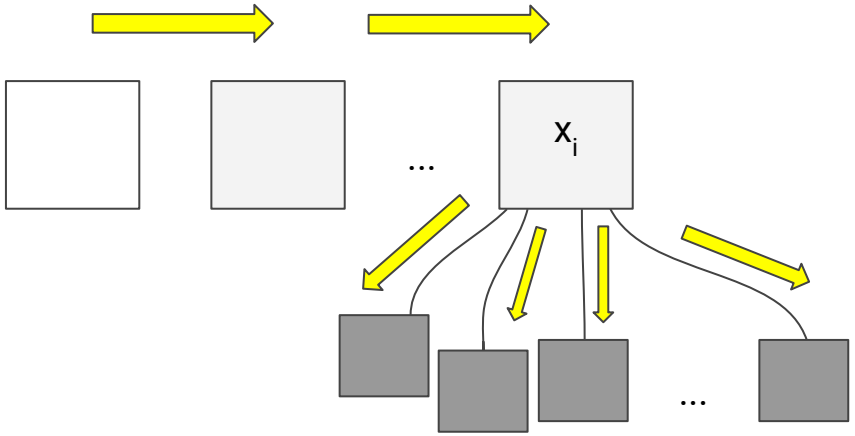
Training Example Generation



$$\text{Total} = N = \kappa * \lambda$$

$$\mathbf{X} = [x_i], \quad i=1...N$$

Algorithm Steps



$$\forall\ a \in A\ ,\ (\ v_{xi}(a),\ p_{xi}(a)\ ) = f_{\theta}(A(x_i,\ a))$$

$$v_i = \max_a(v(s_i,a)+R(A(s_i,a)))$$

$$p_i = \operatorname{argmax}_a\ (v(s_i,a)+R(A(s_i,a)))$$

State - Action Pairs

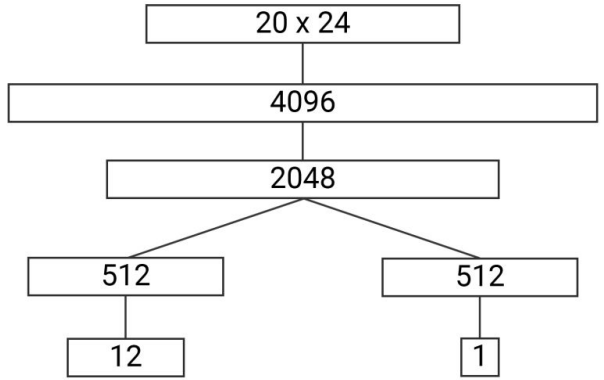
- Supervised learning on the set:
- Treat this as a regression problem
  - ⇒ new parameters **θ'**
- RMSProp Optimizer
  - Gradient descent algorithm with momentum
  - (+) restricts the oscillations in the vertical direction
  - RMS Loss for Value
    - $L(y,\hat{y})=\text{sqrt}\{ (\sum^{N}(y - \hat{y})^2) / n \}$
  - Softmax-Cross-Entropy Loss for Policy
    - $L(y,\hat{y})=-\sum^{N}\{ y(i) \log(\hat{y}(i)) \}$

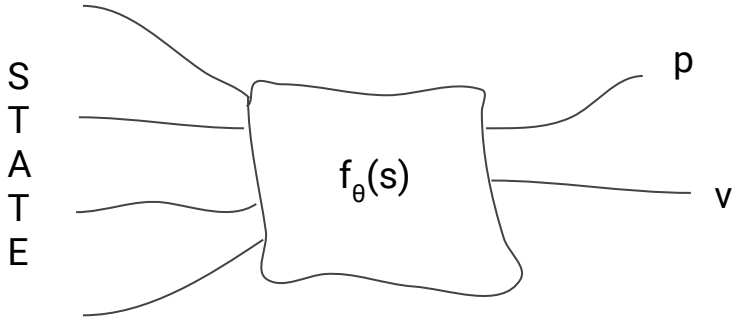
x_1	$target_1$
x_2	$target_2$
...	...
x_n	$target_n$

Neural Network Breakdown

p : vector containing the move probabilities for each ot the 12 possible moves (actions) from that state s

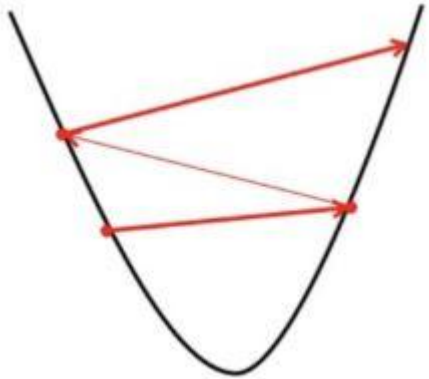
v : single scalar estimating the “goodness” of the state passed. The concrete meaning of a value will be discussed later.





Oh, no! Divergence

- Algorithm did the following:
  - Converged to a degenerate solution
  - Diverged completely



- Solution:
  - Higher weight to samples that are closer to the solved cube
  - **$W(x_i) = 1/D(x_i)$**
  - No divergent behavior after this

Searching Process

Monte-Carlo Tree Search

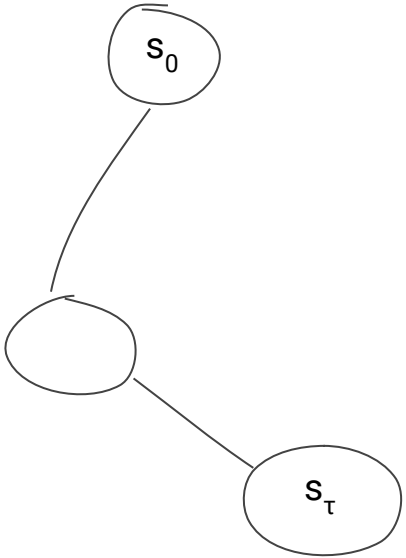
Implementation of asynchronous MCTS augmented with f_θ to solve from s_0

Building the Tree

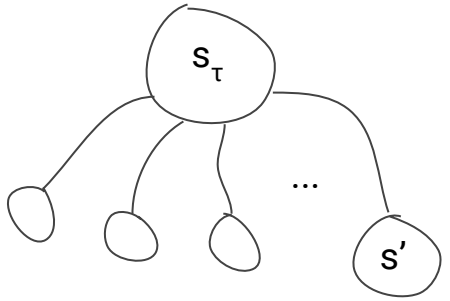
- Initially $T = \{ s_0 \}$
- Simulated traversals until reaching leaf node (s_τ)
- Each state $s \in T$ has a memory attached to it storing:
  - $N_s(a)$: # times action a has been taken from s
  - $W_s(a)$: Maximal value of action a from state s
  - $L_s(a)$: Current virtual loss for action a from state s
  - $P_s(a)$: Prior probability of action a from state s = policy returned by the model

Simulation Phase - Tree Policy

- Simulation start from root -> s_τ (unexplored node - leaf)
  - Follows tree policy while doing so
- Below is the tree policy used (until you bump into a leaf):
  - For each timestamp t, choose action from:
    - $\mathbf{A_t = argmax_a(U_{st}(a) + Q_{st}(a))}$
    - $\mathbf{U_{st}(a) = c * P_{st}(a) \sqrt{(\sum_{a'} N_{st}(a'))/(1+N_{st}(a))}}$
      - c = exploration hyperparameter
    - $\mathbf{Q_{st}(a) = W_{st}(a) - L_{st}(a)}$
      - maximum value returned by the model for all children's states of s under the branch a.



Expansion Phase



- When leaf node is reached (s_τ):
  - Add all children to the tree T - { $A(s_\tau, a), \forall\ a \in A$ }
- For each child in s':
  - Initialize:
    - $W_{s'}=0$,
    - $N_{s'}=0$
    - $\mathbf{P_{s'}= p_{s'}}$ (move towards the direction suggested by the policy calculated by $f_\theta(s')$)
    - $L_{s'}=0$
  - Compute value and policy (u_{st} , p_{st}) = $f_\theta(s_\tau)$
    - Value is backed up on all visited stated in the simulated path

Expansion Phase

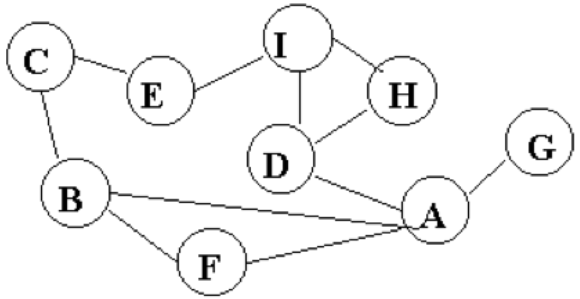
- Update everything in memory as follows:
  - $W_{st}(A_t) \leftarrow \max(W_{st}(A_t), v_{s\tau})$
  - $N_{st} \leftarrow Nst\ (At) + 1$
  - $L_{st}(A_t) \leftarrow L_{st}(A_t) - v$
- Run until:
  - **s_τ is solved state (hopefully)**
  - **out of time**

Only the maximal value encountered along the tree is stored, not the total value - why?

Rubik's Cube is deterministic, not adversarial, we don't need to average our reward when deciding a move (recent paper by same authors using **A*** had a huge improvement)

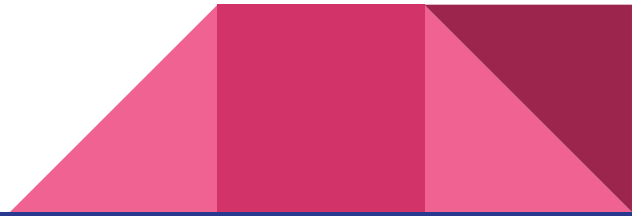
Extract path from tree

- Hopefully s_τ is the solved state
  - Extract tree T and convert to undirected graph
  - Use BFS to find sequence of moves - why?

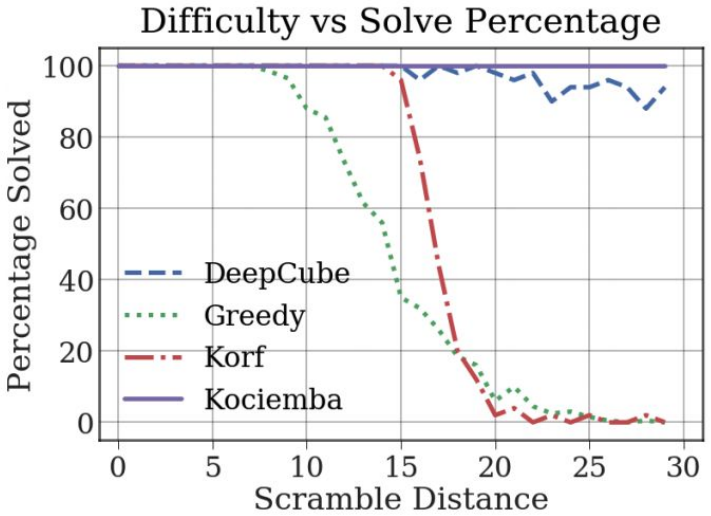


Results

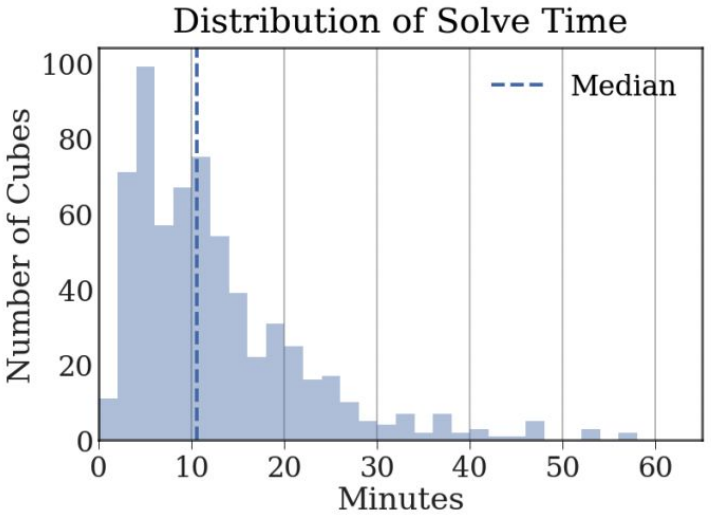
- ADI for 2.000.000 iterations
- Network witnessed ~ 8bn cubes
- Trained for 44h
  - 32-core Intel Xeon E5-2620
  - 3 NVIDIA Titan XP GPUs



Results



(a) Comparison of solve rates among different solvers. Each solver was given 30 minutes to compute a solution for 50 cubes.



(b) DeepCube's Distribution of solve times for the 640 fully scrambled cubes. All cubes were solved within the 60 minute limit.

Results

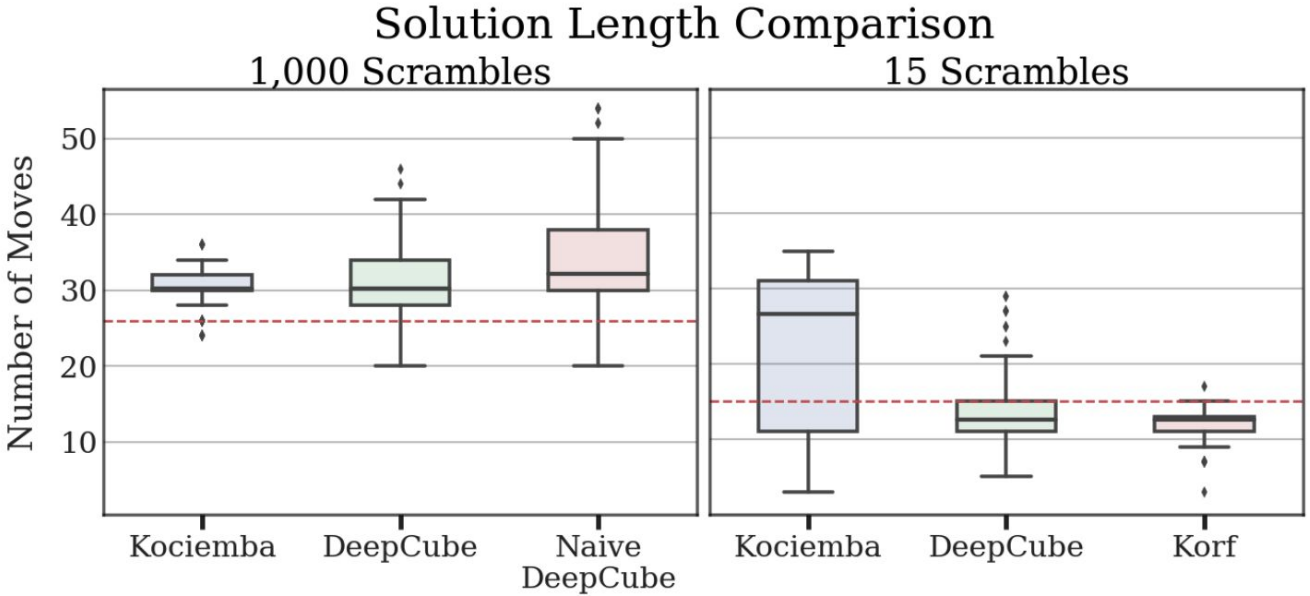
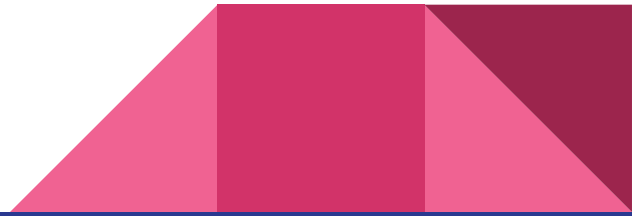


Figure 6: Distribution of solution lengths for DeepCube, Kociemba, and Korf. The left graph features naive DeepCube to evaluate the effect of our shortest path search. The right graph features the Korf optimal solver to evaluate how well DeepCube can find short solutions. The red lines represent the 26 and 15 move upper bound on the left and right respectively.

Question

Visit deepcube.igb.uci.edu and see for yourself this beautiful application of RL. Are there other ways of dealing with sparse rewards in combinatorial RL problems? Where do you suggest Autodidactic Iteration should be implemented (as a policy iteration method) in the future?



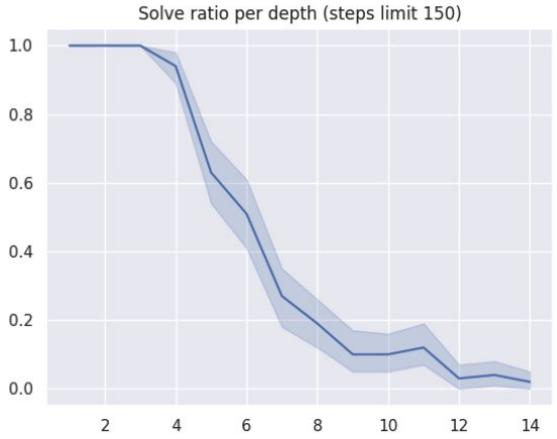
Thank you.

Extra Stuff

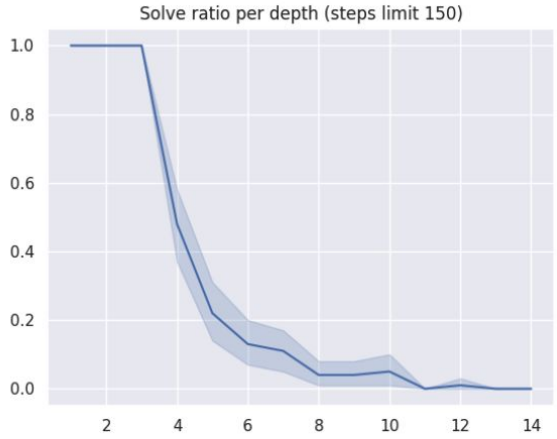
My cheesy entropy strat

$$U_{s_t}(a) = C(P_{s_t} - \frac{V_{s_t} E_{s_t}}{1 + N_{s_t}(a)}) \frac{\sqrt{\sum a' N_{s_t}(a')}}{1 + N_{s_t}(a)}$$

My results when using entropy to guide my agent

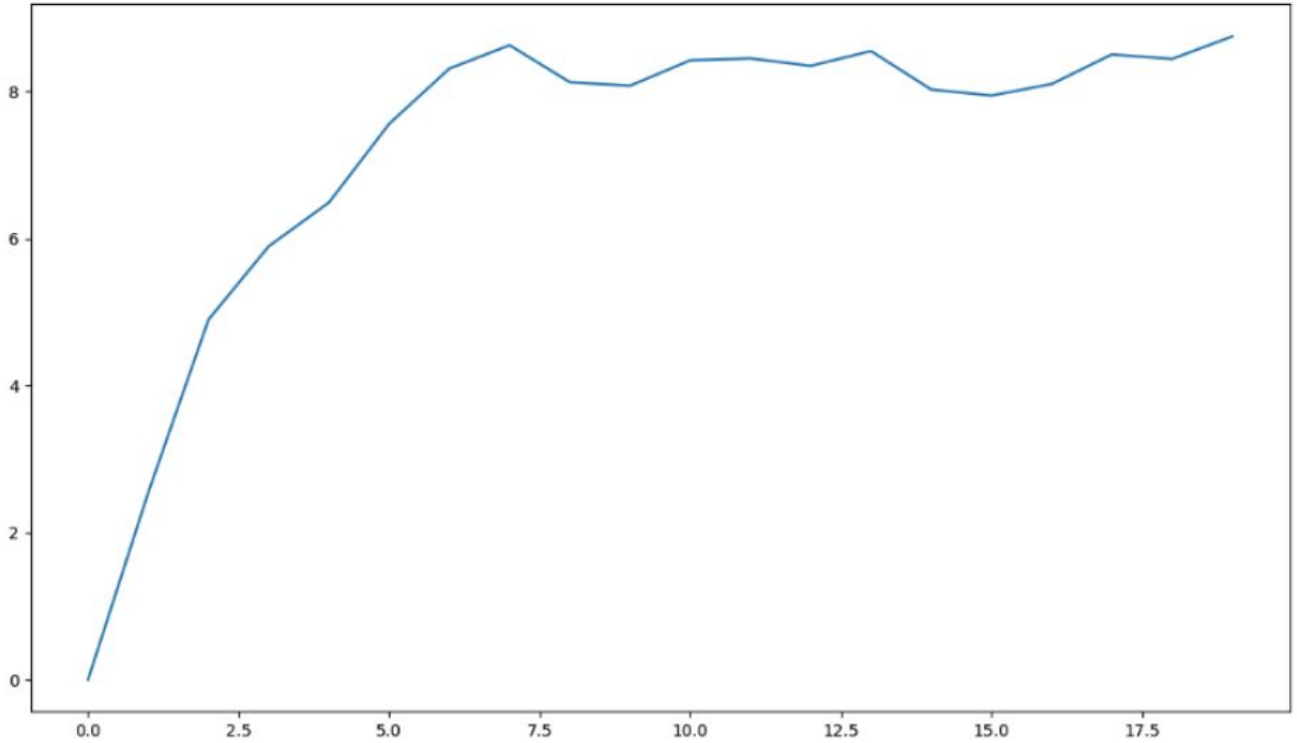


(α') Classic



(β') Classic + Entropy

Entropy behavior with respect to scramble depth



Demo

```
R = 0
L = 1
T = 2
D = 3
F = 4
B = 5
r = 6
l = 7
t = 8
d = 9
f = 10
b = 11
```