ACCELERATING MONTE–CARLO TREE SEARCH WITH OPTIMIZED POSTERIOR POLICIES

KEITH FRANKSTON AND BENJAMIN HOWARD

Abstract. We introduce a recursive AlphaZero-style Monte–Carlo tree search algorithm, "RMCTS". The advantage of RMCTS over AlphaZero's MCTS-UCB [3] is speed. In RMCTS, the search tree is explored in a breadth-first manner, so that network inferences naturally occur in large batches. This significantly reduces the GPU latency cost. We find that RMCTS is often more than 40 times faster than MCTS-UCB when searching a single root state, and about 3 times faster when searching a large batch of root states.

The recursion in RMCTS is based on computing optimized posterior policies at each game state in the search tree, starting from the leaves and working back up to the root. Here we use the posterior policy explored in "Monte–Carlo tree search as regularized policy optimization" [1]. Their posterior policy is the unique policy which maximizes the expected reward given estimated action rewards minus a penalty for diverging from the prior policy.

The tree explored by RMCTS is not defined in an adaptive manner, as it is in MCTS-UCB. Instead, the RMCTS tree is defined by following prior network policies at each node. This is a disadvantage, but the speedup advantage is more significant, and in practice we find that RMCTS-trained networks match the quality of MCTS-UCB-trained networks in roughly one-third of the training time. We include timing and quality comparisons of RMCTS vs. MCTS-UCB for three games: Connect–4, Dots-and-Boxes, and Othello.

1. AlphaZero's MCTS-UCB

AlphaZero [3] is a method to train neural networks to play games at a high level. The main idea is to use Monte-Carlo tree search (MCTS) to explore the game tree and learn an improved value and policy for a given game state, where the exploration is based on prior values and policies from the current network. The network is trained on these improved values and policies, and becomes stronger over time.

In AlphaZero, Monte-Carlo tree search (MCTS-UCB) works roughly as follows (see Algorithm 1 in section 4 for a more precise description). We initialize estimated action values $Q(s,\cdot)$ to zero, and then perform a number of simulations to refine the Q values and obtain a posterior policy. A simulation always begins at the root state, and chooses an action with the maximal UCB value, where the UCB value is defined to be

$$\mathrm{ucb}(s,a) = Q(s,a) + C \cdot \pi_0(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}$$

2020 *Mathematics Subject Classification.* cs.AI, cs.LG.

Key words and phrases. Monte Carlo Tree Search, MCTS, AlphaZero, regularized policy optimization.

where s is the current state, a is an available action from s, $Q(s, a)$ is the estimated reward for taking action a from state s, $C > 0$ is the exploration constant, $N(s, a)$ is the number of times action a has been taken from state s in previous simulations, and finally, $\pi_0(s, \cdot)$ is the prior (network) policy.

We continue picking the action with highest UCB, moving down the exploration tree, until we reach a state s' that is terminal, or which is not yet in the tree. If s' is terminal, then its value is simply the game score. Otherwise, the value of s' is defined to be the network value $v_0(s)$, and s' is added to the tree, initializing $Q(s', \cdot)$ and $N(s', \cdot)$ to zero for all actions from s'. The value of this final state is propagated back up the path to the root using the appropriate sign for the active player; each state along this path updates its Q and N values appropriately.

Updating the Q and N values causes the UCB values to change, and so a different path may be taken in the next simulation. Once $N(s) = \sum_a N(s, a)$ reaches the budgeted number of simulations, the process stops. At this point, the posterior policy $\hat{\pi}$ at a state s is defined to be the normalized visit counts $\hat{\pi}(s, a) = \frac{N(s,a)}{N(s)}$.

Figure 1 illustrates the evolution of UCB values for a simple (one-player) bandit game with two slot arms, starting from a uniform prior policy. Each slot pays out a reward of $+1$ or -1; the probability of $+1$ is a fixed hidden parameter p. For one slot, $p = 0.6$, and for the other, $p = 0.4$. The inferior choice ($p = 0.4$) is never abandoned; this action is chosen $\Theta(1/\sqrt{N})$ of the time, where N is the total number of simulations. As $N \to \infty$, the two UCB values get closer and closer together, approaching $0.2 + \Theta(1/\sqrt{N})$.

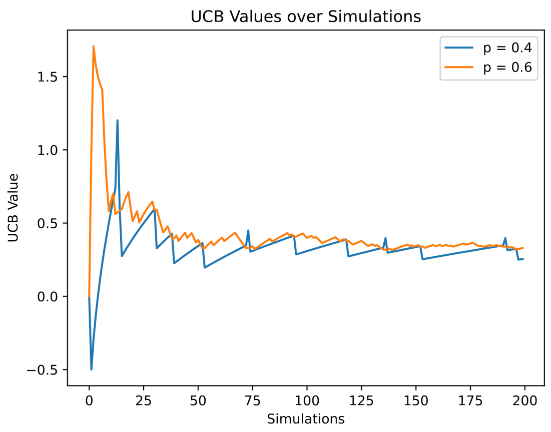

FIGURE 1. Illustration of UCB values over 200 simulations in bandit game with two slot arms.

2. REGULARIZED POLICY OPTIMIZATION

An alternative to $\hat{\pi}$ was explored in "Monte-Carlo tree search as regularized policy optimization" [1]. They define the optimized posterior policy $\bar{\pi}(s, \cdot)$ to be the one maximizing the expected reward (assuming estimated rewards $Q(s, a)$) minus a penalty for diverging from the prior policy $\pi_0(s, \cdot)$. Specifically, $\bar{\pi}(s, \cdot)$ is the unique distribution on actions at game state s which maximizes

$$\sum_a \bar{\pi}(s,a)Q(s,a) - \frac{C}{\sqrt{N(s)}} \operatorname{KL-Div}(\pi_0(s,\cdot) \mid\mid \bar{\pi}(s,\cdot))$$

where KL-Div is the Kullback-Leibler divergence, and $C > 0$ is the same exploration constant as used in MCTS-UCB. They show that $\bar{\pi}$ agrees asymptotically to $\hat{\pi}$ (normalized visit counts) as the number of MCTS simulations approaches infinity, but argue that $\bar{\pi}$ is superior to $\hat{\pi}$ when the number of simulations is small. They suggest replacing $\hat{\pi}$ with $\bar{\pi}$ in the AlphaZero algorithm, where the Q-values are computed using the original MCTS-UCB variant of AlphaZero.

We note that the optimized posterior policy $\bar{\pi}$ has an interesting interpretation: it is essentially the policy which forces the UCB values of all actions to be equal. Recall that the UCB value is defined as

$$\operatorname{ucb}(s,a) = Q(s,a) + C \cdot \pi_0(s,a) \cdot \frac{\sqrt{N(s)}}{1 + N(s,a)}.$$

Now suppose that we have finished all the simulations at state s, and $\hat{\pi}(s,a) = \frac{N(s,a)}{N(s)}$ is the normalized visit counts. For simplicity, let us drop the "1+" in the denominator of the exploration term. Now we can rewrite the UCB value as

$$\begin{aligned}\operatorname{ucb}(s,a) &= Q(s,a) + C \cdot \pi_0(s,a) \cdot \frac{\sqrt{N(s)}}{\hat{\pi}(s,a)N(s)} \\ &= Q(s,a) + \frac{C}{\sqrt{N(s)}} \cdot \frac{\pi_0(s,a)}{\hat{\pi}(s,a)}.\end{aligned}$$

Now replace $\hat{\pi}(s,a)$ with a variable $\bar{\pi}(s,a)$, and consider the problem of maximizing the objective function

$$F(\bar{\pi}(s,\cdot)) = \sum_a \bar{\pi}(s,a)Q(s,a) - \frac{C}{\sqrt{N(s)}} \operatorname{KL-Div}(\pi_0(s,\cdot) \mid\mid \bar{\pi}(s,\cdot)),$$

subject to the constraint that $\sum_a \bar{\pi}(s,a) = 1$. A simple calculation shows that $\frac{\partial F}{\partial \bar{\pi}(s,a)} = \operatorname{ucb}(s,a)$. On the other hand, the constraint function $\sum_a \bar{\pi}(s,a) = 1$ has partial derivative 1 with respect to each variable $\bar{\pi}(s,a)$. Hence the method of Lagrange multipliers tells us that the optimum $\bar{\pi}(s,\cdot)$ can only occur where all the UCB values $\operatorname{ucb}(s,a)$ are equal. From Figure 1 one can see that the UCB values of MCTS-UCB approach each other over time; this optimized posterior policy $\bar{\pi}$ forces them to be equal.

If u is the common UCB value for the optimal posterior policy $\bar{\pi}(s,\cdot)$, then we have

$$\bar{\pi}(s,a) = \frac{C}{\sqrt{N(s)}} \frac{\pi_0(s,a)}{u - Q(s,a)}.$$

In particular, we are looking for the unique value of $u > \max_a Q(s,a)$ such that

$$\sum_a \frac{C}{\sqrt{N(s)}} \frac{\pi_0(s,a)}{u - Q(s,a)} = 1.$$

This value certainly exists, since as $u \to \max_a Q(s,a)^+$, the left-hand side approaches $+\infty$, whereas as $u \to +\infty$, the left-hand side approaches 0.

The optimal posterior policy $\bar{\pi}$ can be computed efficiently using Newton's method (cf. Algorithm 4 in section 4). In this algorithm, the function f is convex and since we begin with the initial value on the appropriate side of the solution, Newton's method is guaranteed to converge.

3. RMCTS

An important feature of the optimized posterior policy $\bar{\pi}$ is that it can be computed locally at each game state s, using only the estimated action values $Q(s,\cdot)$ and the prior policy $\pi_0(s,\cdot)$. It does not require any details about the rest of the search tree. This is what permits us to define a recursive alternative to MCTS-UCB, which we call RMCTS.

Recall that [1] suggests replacing the posterior policy $\hat{\pi}$ with the optimized posterior policy $\bar{\pi}$, where the estimated Q values are gotten from the original MCTS-UCB variant of AlphaZero. By contrast, RMCTS completely redefines MCTS itself, by using optimized policies not only at the root state, but throughout the tree. The Q-values are computed in a recursive manner, where the value of nodes s' in the search tree are approximated (recursively) by computing optimized posterior policies below s'. See Algorithm 2 in section 4 for a succinct description of RMCTS.

The search tree is generated by following prior network policies at each node (in particular it is not defined by UCB values). Each node s in the tree consumes one simulation[1], and then awards the remaining simulations to its child actions according to the prior policy $\pi_0(s,\cdot)$. Thus if a state s is afforded $N(s)$ simulations, then one simulation is used to acquire the prior policy $\pi_0(s,\cdot)$ and value $v_0(s)$ from the network, leaving $N(s)-1$ simulations to be distributed among the available actions from s. The number of simulations $N(s,a)$ assigned to action a (cf. Algorithm 3) has expectation $\mathbb{E}[N(s,a)] = \pi_0(s,a)(N(s)-1)$, where π_0 is the prior network policy at state s. As in MCTS-UCB, if s' is a nonterminal leaf (afforded one simulation), then its value is defined to be the network value $v_0(s')$. If s' is a terminal state, then its value is simply the game score. See section 5 for a simple example illustrating RMCTS.

3.1. **Implementing RMCTS efficiently.**

The description of RMCTS in section 4 is mathematically precise, but it is not efficient to implement this way, where the function calls itself recursively. See `https://github.com/bhoward73/rmcts` for an efficient C implementation of RMCTS. This efficient implementation of Algorithm 2 works iteratively, exploring the search tree in a breadth-first manner. All nodes at the same depth form one large batch of GPU inferences, and so the GPU latency cost is largely mitigated. Contiguous memory is pre-allocated for all the nodes (and relevant data) for the search tree, improving cache performance.

3.2. **Timings and quality comparisons.**

The timings we report in section 6 are based on the efficient implementation of RMCTS described above. We compare the quality of RMCTS and MCTS-UCB in section 7, by pitting them against each other in three games: Connect-4, Dots-and-Boxes, and Othello. In this contest, both RMCTS and MCTS-UCB use the same neural network for priors.

[1]Since GPU inference is the bottleneck, and we need such an inference at every node of the tree, we define the simulation count to drop by one at every node.

4. Algorithm Descriptions

In this section we give precise descriptions of MCTS-UCB (Algorithm 1) and RMCTS (Algorithm 2).

Algorithm 1: MCTS-UCB

```
Input: State $s$, number of simulations $N$, exploration constant $C$
Output: New policy $\hat{\pi}$ at root state $s$
1  Given state $t$, let $\mathrm{sgn}(t) = +1$ if player 1 to move in state $t$, else $-1$;
2  visited $= \emptyset$;
3  while $N > 0$ do
4      $t \leftarrow s$;
5      path $= []$;
6      while $t$ is in visited and $t$ is nonterminal do
7          Select action $a$ that maximizes
           $\mathrm{ucb}(t, a) = Q(t, a) + C \cdot \pi_0(t, a) \frac{\sqrt{\sum_b N(t,b)}}{1+N(t,a)}$;
8          Append $(t, a)$ to path;
9          $t \leftarrow$ state reached by taking action $a$ from state $t$;
10     if $t$ is nonterminal then
11         Add $t$ to visited;
12         $Q(t, a) \leftarrow 0$ for all actions $a$;
13         $N(t, a) \leftarrow 0$ for all actions $a$;
14         Acquire priors $v_0(t)$ and $\pi_0(t, \cdot)$ from neural network;
15         $v \leftarrow \mathrm{sgn}(t) v_0(t)$ (appropriate sign relative to player 1);
16     else
17         $v \leftarrow$ score of terminal state $t$, relative to player 1;
18     foreach $(t, a)$ in path do
19         $N(t, a) \leftarrow N(t, a) + 1$;
20         $Q(t, a) \leftarrow Q(t, a) + \frac{\mathrm{sgn}(t)v - Q(t,a)}{N(t,a)}$;
21     $N \leftarrow N - 1$;
22 foreach action $a$ do
23     $\hat{\pi}(a) \leftarrow \frac{N(s,a)}{\sum_b N(s,b)}$;
24 return $\hat{\pi}$
```

```
Algorithm 2: RMCTS (deterministic, two-player zero-sum)
   Input: State $s$, number of simulations $N$, exploration constant $C$
   Output: Estimated value $\bar{v}$ and policy $\bar{\pi}$ at root state $s$
1  if $s$ is terminal then
2      return score of s, NULL policy;
3  Let $\mathrm{sgn}(s,t) = +1$ if same player is active for states $s$ and $t$, else $-1$;
4  Acquire priors $v_0(s)$ and $\pi_0(s,\cdot)$ from neural network;
5  $N(s,\cdot) \leftarrow$ ASSIGN-SIMULATIONS$(s, N-1, \pi_0(s,\cdot))$; // Algorithm 3
6  $A \leftarrow \{a : N(s,a) > 0\}$;
7  foreach action $a \notin A$ do
8      $\bar{\pi}(s,a) \leftarrow 0$;
9  foreach action $a \in A$ do
10     let $t$ be the state reached by taking action $a$ from state $s$;
11     $v_t, \pi_t \leftarrow$ RMCTS$(t, N(s,a), C)$;
12     $Q(s,a) \leftarrow \mathrm{sgn}(s,t)\, v_t$;
13 Let $Q', \pi_0'$ be the restriction of $Q(s,\cdot)$ and $\pi_0(s,\cdot)$ to actions in $A$;
14 $\pi' \leftarrow$ POLICY-OPTIMIZATION$(Q', \pi_0', N-1, C)$; // Algorithm 4
15 foreach action $a \in A$ do
16     $\bar{\pi}(s,a) \leftarrow \pi'(a)$;
17 $\bar{v} \leftarrow \frac{1}{N} \cdot v_0(s) + \frac{N-1}{N} \cdot \sum_{a \in A} Q(s,a) \cdot \bar{\pi}(s,a)$;
18 return $\bar{v}, \bar{\pi}$
```

```
Algorithm 3: ASSIGN-SIMULATIONS
   Input: State $s$, number of simulations $N$, prior policy $\pi_0(s,\cdot)$
   Output: Number of simulations $N(s,a)$ assigned to each action $a$
1  Put all potential actions in some arbitrary order $a_1, a_2, \ldots, a_n$;
2  $t_0 \leftarrow 0$;
3  for $i = 1, 2, \ldots, n$ do
4      $t_i \leftarrow N \sum_{j \le i} \pi_0(s, a_j)$;
5  Generate a uniformly random number $x$ in $[0,1)$;
6  for $i = 1, 2, \ldots, n$ do
7      $N(s,a_i) \leftarrow \#\{k \in \mathbb{Z} : t_{i-1} \le x + k < t_i\}$;
8  return $N(s,a)$ for each action $a$
```

The subroutine Algorithm 3 (ASSIGN-SIMULATIONS) randomly distributes a total of N simulations to actions according to the prior policy $\pi_0(s,\cdot)$. Action a is assigned $N(s,a)$ simulations. The expectation $\mathbb{E}[N(s,a)] = \pi_0(s,a)N$, and $\lfloor \pi_0(s,a)N \rfloor \le N(s,a) \le \lceil \pi_0(s,a)N \rceil$. We note that, in particular, every action a is guaranteed to receive at least $\lfloor \pi_0(s,a)N \rfloor$ simulations. Finally, we have $\sum_a N(s,a) = N$.

Algorithm 4: POLICY-OPTIMIZATION

```
    Input: estimated action rewards $Q$, prior policy $\pi_0$, number of simulations
           $N$, exploration constant $C$
    Output: Updated policy $\bar{\pi}$ as defined in [1], maximizing
            $\sum_a \bar{\pi}(a)Q(a) - \frac{C}{\sqrt{N}}$ KL-Div$(\pi_0 \,||\, \bar{\pi})$
 1  $\epsilon \leftarrow 10^{-10}$;
 2  $\lambda \leftarrow C/\sqrt{N}$;
 3  Define $f(u) \leftarrow -1 + \lambda \sum_a \frac{\pi_0(a)}{u - Q(a)}$;
 4  $u \leftarrow \max_a Q(a) + \lambda\pi_0(a)$;
 5  while $f(u) > \epsilon$ do
 6      $u \leftarrow u - \frac{f(u)}{f'(u)}$ // Newton's method always converges here
 7  foreach action $a$ do
 8      $\bar{\pi}(a) \leftarrow \lambda \frac{\pi_0(a)}{u - Q(a)}$;
 9  Normalize $\bar{\pi}$ so that $\sum_a \bar{\pi}(a) = 1$;
10  return $\bar{\pi}$
```

5. A simple example

Consider the following toy-sized one-player game illustrated in Figure 2, where the game tree is a binary tree with only two nonterminal states s and t. The available actions are ℓ (left) and r (right). Starting from the root state s, if we go left, we reach a terminal state with value 1. If instead we go right, we reach the nonterminal state t. From t, if we go left, we reach a terminal state with value -3. If we go right from t, we reach a terminal state with value 2.

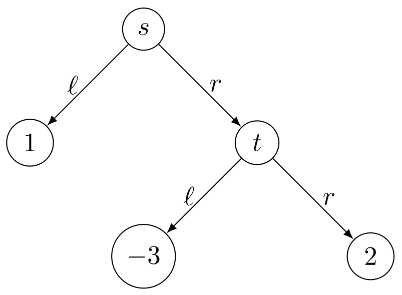

Figure 2. Clearly the best action from state s in this one-player game is to go right, reaching state t, where we should again choose to go right, obtaining the maximal reward of 2.

Suppose that we are afforded $N = 1003$ simulations at the root state s. Initially we have no strong opinions and our prior policy is uniform, and our prior value is zero on all nonterminal states. Let's set our exploration constant to $C = 1$.

We spend one simulation on s itself, for which our prior value is $v_0(s) = 0$, leaving us with 1002 simulations to distribute among the two actions ℓ and r from s. Since the prior policy is uniform we assign $N(s, \ell) = 501$ simulations to action ℓ and also $N(s, r) = 501$ simulations to action r. Since the left action ℓ from s reaches a terminal state with value 1, we know the exact value $Q(s, \ell) = 1$ for the left action. For the right action r from s, we first consume one simulation to obtain the uniform prior policy and zero prior value at state t. We assign $N(t, \ell) = 250$ simulations to action ℓ and $N(t, r) = 250$ simulations to action r from state t.

Now that the search tree and simulation counts have been defined, we now turn to the recursive computation of Q values. The optimized posterior policy at state t (Algorithm 4) is approximately $\bar{\pi}(t, \ell) = 0.00445$ and $\bar{\pi}(t, r) = 0.996$. Giving one vote to the prior value $v_0(t) = 0$, our new estimated value $\bar{v}(t) = Q(s, r)$ is

$$Q(s, r) = \bar{v}(t) = \frac{1}{501} \cdot 0 + \frac{500}{501} \cdot (-3 \cdot 0.00445 + 2 \cdot 0.996) \approx 1.98.$$

Now that all estimated Q values at state s are known, we can compute the optimized posterior policy at s. The optimized posterior policy at the root state s is computed (Algorithm 4) to be approximately $\bar{\pi}(s, \ell) = 0.016$ and $\bar{\pi}(s, r) = 0.984$. Thus, our final estimated value at the root state s is

$$\bar{v}(s) = \frac{1}{1003} \cdot 0 + \frac{1002}{1003} \cdot (1 \cdot 0.016 + 1.98 \cdot 0.984) \approx 1.96.$$

Hence the posterior value of s is approximately 1.96, and the posterior policy at s strongly favors going right with a probability of 98.4%.

Note that it was very important to use recursion to define the Q values. If we had assigned the Q values to be the expected action values following the prior policy π_0 (no recursion), then our new posterior policy $\bar{\pi}$ would have favored going left from s, since going left yields a reward of 1, whereas the expected value of going right is $(-3 + 2)/2 = -0.5 < 1$.

6. Timings for MCTS-UCB vs RMCTS for three games

In this section we give timing comparisons of MCTS-UCB vs RMCTS for three games: Connect-4, Dots-and-Boxes, and Othello. All timings use TensorRT to optimize the neural network inferences. We ran these tests on a desktop computer which has an NVIDIA RTX 3080 GPU, and an Intel i7-10700K CPU. In the cases of Dots-and-Boxes and Othello, the network was a ResNet with 8 residual blocks and 48 channels. For Connect-4, the network was a ResNet with 8 residual blocks and 64 channels. In all cases the kernel size was 3×3.

We first consider the case where we are timing MCTS for a single root state. This situation applies when we (human) play against a pre-trained AI opponent, and we want to supplement the AI network with MCTS at each move. Here RMCTS has the greatest speed advantage over MCTS-UCB.

N	32	64	128	256	512	1024	2048
MCTS-UCB	13 ms	25 ms	49 ms	98 ms	200 ms	390 ms	790 ms
RMCTS	1.9 ms	2.5 ms	3.2 ms	4.2 ms	5.8 ms	8.9 ms	14 ms
speedup	6.8×	10×	16×	24×	34×	44×	57×

Table 1. Othello timings, one root state.

N	32	64	128	256	512	1024	2048
MCTS-UCB	13 ms	25 ms	47 ms	93 ms	190 ms	370 ms	730 ms
RMCTS	3.3 ms	4.7 ms	5.7 ms	6.7 ms	7.3 ms	8.0 ms	10 ms
speedup	3.9×	5.2×	8.5×	15×	26×	48×	77×

Table 2. Dots-and-Boxes timings, one root state.

N	32	64	128	256	512	1024	2048
MCTS-UCB	18 ms	35 ms	70 ms	150 ms	280 ms	550 ms	1100 ms
RMCTS	3.8 ms	4.9 ms	6.8 ms	9.2 ms	9.7 ms	12 ms	15 ms
speedup	4.8×	7.2×	10×	16×	28×	45×	74×

Table 3. Connect-4 timings, one root state.

We next consider the case when we are computing MCTS for a batch of root states (trees) "at once." This situation applies when we are using MCTS to generate rollouts for training the neural network. Here the latency cost of network inferences is largely mitigated for both MCTS-UCB and RMCTS, since the network inferences are done in batches in both cases. In MCTS-UCB, when searching a batch of root states, we move from one state to next whenever a necessary inference prevents us from continuing the computation at the given state. Once we've passed through all states in the batch, then we wait for responses from the network before continuing. The latency cost is mitigated in this way, but RMCTS maintains a significant speed advantage, because the batch sizes for RMCTS are much larger, since an RMCTS batch consists of all nodes at a given depth across all search trees.

N	32	64	128	256	512	1024	2048
MCTS-UCB	0.58 ms	1.2 ms	2.5 ms	5.3 ms	12 ms	27 ms	63 ms
RMCTS	0.19 ms	0.33 ms	0.60 ms	1.1 ms	2.2 ms	4.2 ms	8.5 ms
speedup	3.1×	3.5×	4.1×	4.7×	5.5×	6.6×	7.3×

Table 4. Othello, 64 root states, average time per root state.

N	32	64	128	256	512	1024	2048
MCTS-UCB	0.54 ms	1.1 ms	2.2 ms	4.6 ms	9.9 ms	23 ms	51 ms
RMCTS	0.15 ms	0.22 ms	0.33 ms	0.57 ms	0.92 ms	1.7 ms	2.8 ms
speedup	3.5×	4.9×	6.7×	8.2×	10.7×	13.7×	18.1×

Table 5. Dots-and-Boxes, 64 root states, average time per root state.

N	32	64	128	256	512	1024	2048
MCTS-UCB	0.40 ms	0.80 ms	1.5 ms	3.1 ms	6.1 ms	12 ms	25 ms
RMCTS	0.26 ms	0.36 ms	0.53 ms	0.82 ms	1.4 ms	2.5 ms	4.5 ms
speedup	1.5×	2.2×	2.9×	3.7×	4.4×	4.8×	5.4×

Table 6. Connect-4, 64 root states, average time per root state.

7. Comparing the quality of RMCTS vs MCTS-UCB

We find that RMCTS is at a small disadvantage if the number of simulations is equal. Most likely the reason for this is that MCTS-UCB creates the search tree in an adaptive manner, whereas RMCTS creates the search tree in a non-adaptive manner. Nevertheless, the speed advantage of RMCTS more than makes up for this disadvantage. See Table 7 for an example of this in Othello. It seems a general rule of thumb that RMCTS wins games against MCTS-UCB when the number of simulations for RMCTS is twice that of MCTS-UCB, but RMCTS still takes far less time than MCTS-UCB even though it is using twice the number of simulations.

Table 7 shows the results of RMCTS vs MCTS-UCB in 64 games of Othello (32 played as first player, and 32 as second player). In both cases, MCTS-UCB had $N = 256$ simulations and required about 2.3 seconds per game. In the first batch of 64 games, RMCTS also had $N = 256$ simulations, but generally lost with a mean score (checker difference) of -4.0. The time required by RMCTS per game, however, was only 135 milliseconds. In the second set of 64 games, RMCTS was given $N = 512$ simulations, and this time it won with a mean score of 7.2. The mean time per game was 178 milliseconds in this case – still far less than the MCTS-UCB time of 2.3 seconds.

N for RMCTS	mean score	mean time per game	speedup over MCTS-UCB
256	-4.0	135 milliseconds	17×
512	7.2	178 milliseconds	13×

Table 7. Pitting RMCTS vs MCTS-UCB in 64 games of Othello. Doubling the number of simulations for RMCTS gives it both a score advantage and speed advantage.

To compare timings for training with 64 games at once, the average RMCTS time with $N = 512$ simulations is about 2.2 milliseconds, whereas the average time for MCTS-UCB with $N = 256$ simulations is about 5.3 milliseconds. Hence RMCTS has the advantage both in quality and time, with a speedup factor of 2.4×. Equating their performance, the speedup increases to above 3×. This is the kind of comparison that matters most for training, and indeed this factor of 3× speedup in training time is what we observed in practice.

8. What next?

In the near future we plan to test an adaptive variant of RMCTS. We do k re-explorations at the same root state, continually refining the policies at all explored nodes by taking a weighted average of the prior policy and the posterior policy from the most recent exploration. With each iteration, the subtree goes deeper into branches favored by the posterior policies. The nodes that were previously explored would not count against the simulation budget since we would store their values and policies from the previous explorations. Thus the overall procedure becomes adaptive, and would likely outperform the non-adaptive version.

We emphasize that RMCTS is not limited to the specific optimal posterior policy of [1] (Algorithm 4). Other variants of optimized posterior policies can be considered, for example we can replace the Kullback-Leibler divergence with other divergence measures. It would be interesting to see how well these variants perform. For example, if we reverse the order of π_0 and $\bar{\pi}$ in the Kullback-Leibler divergence: i.e. $\bar{\pi}$ maximizes

$$\sum_a \bar{\pi}(s,a)Q(s,a) - \frac{C}{\sqrt{N(s)}}\,\text{KL-Div}(\bar{\pi} \mid\mid \pi_0),$$

then $\bar{\pi}(s,a) \propto \pi_0(s,a)\exp\left(\frac{\sqrt{N(s)}}{C}Q(s,a)\right)$. This is an appealing formula, and easy to calculate; but it probably punishes the least-favored actions too much.

In general, RMCTS makes sense for reasonable variants of Algorithm 4, where the optimized posterior policy can be computed efficiently, and is computed based only on the estimated Q-values, prior policy, number of simulations, and exploration constant.

Throughout this paper we have only talked about AlphaZero, and not mentioned its successor MuZero [2]. However, RMCTS also applies to MuZero, since the search procedure of MuZero is once again MCTS-UCB, excepting that it is done in the simulated latent space of the learned model.

9. Acknowledgements

We thank Tom Sznigir, James Barker, Timothy Chow, Emma Cohen, Ryan Eberhart, Stephen Fischer, Stephen Boyack, Ross Parker, and Lawren Smithline (all of IDA/CCR-Princeton) for helpful discussions and comments. We thank Laurent Sartran and Thomas Hubert of Google/DeepMind for helpful discussions at the Institute for Advanced Study in Princeton, NJ, during the Spring of 2025. We thank Akshay Venkatesh for inviting the second author to attend the DeepMind presentation at IAS.

References

[1] Jean-Bastien Grill, Florent Altché, Yunhao Tang, Thomas Hubert, Michal Valko, Ioannis Antonoglou, and Rémi Munos. Monte-carlo tree search as regularized policy optimization. In *International Conference on Machine Learning*, pages 3769–3778. PMLR, 2020.

[2] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[3] 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. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

Center for Communications Research, Institute for Defense Analyses, Princeton, NJ 08540, USA
Email address: `k.frankston@fastmail.com`
Email address: `k.frankston@idaccr.org`

Center for Communications Research, Institute for Defense Analyses, Princeton, NJ 08540, USA
Email address: `bhoward73@gmail.com`
Email address: `bjhowa3@idaccr.org`