Game tree search using realization probabilities and policy refinement

Graham Jones

2021-02-14, March 25, 2021

email: art@gjones.name website: www.indriid.com

Abstract

The realization probability of a node in a game tree represents the probability that the moves leading to the node will actually be played, and is the product of the probabilities of the moves along the path from root to node. The tree search algorithm used by AlphaZero chooses the next node to expand by using a score based on an upper confidence bound. This paper describes a tree search algorithm which is similar, except that it uses a score based on realization probabilities instead. The focus here is on game play, given a function which provides an evaluation of the game state, and a policy, which provides a probability for each move. In the method introduced here, a function similar to softmax is used to convert evaluations into a policy at each node that has any evaluated children. These derived policies are used to refine the initial policies. As a proof of concept, the performance of algorithm is compared to the upper confidence algorithm on simulated data for non-existent games. The simulations are designed to produce game trees, estimated evaluations, and estimated policies with similar statistical properties to those of real games. On this data, the proposed algorithm shows superior performance. [chess, shogi, go, Monte Carlo tree search, upper confidence trees]

1 Introduction

This paper describes a tree search algorithm for games such as chess, shogi, and go. Traditional approaches use alpha-beta search. However, most of the successful ones, such as the chess program Stockfish (`https://stockfishchess.org/`), use a variety of other heuristics to the limit the breadth of the search. Elnaggar et al. (2014) provides a general survey of such methods. Chess has a branching factor of around 30, so a search tree using alpha-beta search alone will have a branching factor of at least 5 or 6. Actually the search trees in Stockfish have a branching factor around 2. The heuristics are often game-specific.

The tree search algorithm used by AlphaZero Silver et al. (2018) is simpler and more generic, with only a few parameters to be adjusted for particular games and evaluation functions. We will refer to this algorithm as the PUCT algorithm (policy + upper confidence tree) instead of MCTS (Monte Carlo tree search). The algorithm is deterministic, and the 'Monte Carlo' is liable to cause confusion. PUCT uses both evaluations, which are estimates of the expected game score given a game state, and a policy vector which estimates the probability that each move from a given state is the optimal move. PUCT updates the evaluations as the tree grows, and uses these updated evaluations, plus another term to encourage exploration, when choosing the next node to expand. The combination can be interpreted as 'exploitation + exploration', or as an upper confidence bound for the evaluation.

The algorithm presented here has much in common with PUCT, but uses realization probabilities, and represents an attempt to understand the problem as one of statistical inference. The realization probability of a node represents the probability that the moves leading to the node will actually be played (Tsuruoka et al., 2002). The root of the tree has realization probability equal to 1. There are then transitional probabilities at each node which represent the probabilities of the moves from that node. By multiplying the transitional probabilities along the path from the root, the realization probability of any node can be found. The list of transitional probabilities for a node is essentially the same thing as a policy vector, and in this paper we regard them as interchangeable. We will refer to the evaluations and policies that are provided to the algorithm as 'static evaluations' and 'static policies' when there is a need to distinguish them from updated values.

The static policies are our initial estimate of the transitional probabilities. The basic approach described here is to refine these using evaluations, and to use their current values to choose which node to expand next. When a node is expanded, the evaluations are updated in the same way as PUCT. Evaluations can be converted into a policy using a softmax, for example. The softmax can be improved upon, as we will see in section 2.2. We also take into account the uncertainty in the policy and evaluation estimates, using the concept of effective sample sizes (ESSs). The guiding principle behind the choice of node to expand is to make the number of times each node in the tree has been visited approximately proportional to its realization probability. Section 2.3 describes some ways of doing this. We call the proposed algorithm ESPRIT (Effective Sample size and Policy Refinement In Trees).

As a proof of concept, we compare this algorithm against PUCT on simulated trees for a non-existent game. We take some trouble to simulate trees, estimated evaluations, and estimated policies that have similar statistical properties to those in real games. The main advantage of simulations is that the true evaluations and best moves are known, so the regret can be calculated exactly. The main disadvantage is that success on such data may fail to translate to real games and evaluation functions.

An important statistical property of an evaluation function is the distribution of its errors. One can use existing chess engines to compare their evaluations for long and short times. Taking the slow evaluations as an approximation to the truth, the shape of the noise distribution of the fast evaluations can be estimated. Naturally the results depend on the engines and the board positions chosen, but it is a common observation that a few evaluations, say one in a thousand, will be more than 6 standard deviations away from the true evaluation. This should not be surprising, nor be viewed as a special feature of particular games and evaluation functions. Real-world data is often like this and can modeled with appropriate probability distributions (Lange et al., 1989; Ripley, 1996, p39).

Realization probabilities have been used in game tree search before, especially for shogi. In most methods, the realization probabilities are used to limit depth in alpha-beta search. They were introduced in Tsuruoka et al. (2002), where the transitional probabilities are derived from games played by professional shogi players. There is some further work along these lines, including Winands and Björnsson (2008) and Kimura et al. (2011).

One approach, developed in Kirii et al. (2017) and Igarashi et al. (2019) uses realization probabilities on their own for game tree search. This method is designed for use in the context of learning. It uses softmax to convert evaluations to transitional probabilities. The search iteratively chooses a move by sampling from the transitional probabilities at each node. This means the next node to be expanded is chosen with a probability proportional to its realization probability.

There have been various other attempts to improve the PUCT algorithm. Some recent ones are Grill et al. (2020), which we discuss further in section 2.4, Dam et al. (2020) and Hamrick et al. (2020).

2 Search algorithms

We consider a node s with M moves $1, \dots, M$. We can also regard s as a game state, for example a board position in a chess game. A move m may refer to a child node or to an unexpanded node. Each node is assumed to have a static evaluation in $[0, 1]$ which is an estimate of the expected game score and a static policy of form $p = (p_1, \dots, p_M)$ where p_m is an estimate of the probability that m is the best move. The number of static evaluations at or below a node m is n_m. The recursive evaluation of node m is v_m from the point of view of the player at node m. If $n_m > 0$, this is the mean of the static evaluations at or below node m, each converted to the point of view of the player at node m. When n_m is zero, v_m is set to zero too. For both PUCT and ESPRIT, other choices for v_m when $n_m = 0$ can be considered, but I am not aware of anything that is clearly better for either algorithm.

2.1 PUCT

The PUCT formula used here is based on my interpretation of file `pseudocode.py` in the supplementary information to Silver et al. (2018). Note that $n_s = 1 + \sum_m n_m$, that is, the number of visits to s, which differs slightly from some other descriptions in the literature, which omit the 1. The formula is used to provide values $(v_m + u_m)$ for guiding the search, where u_m is given, for each move m from a node s, by

$$u_m = C \frac{\sqrt{n_s}}{1 + n_m} p_m \tag{1}$$

where

$$C = c_0 + \log\left(\frac{1 + n_s + c_1}{c_1}\right) \tag{2}$$

and where c_0 and c_1 are parameters. AlphaZero used $c_0 = 1.25$ and $c_1 = 19652$ so C is a slowly increasing function of n_s. The next node to be expanded is found iteratively from the root, choosing the move at each node which maximizes $(v_m + u_m)$, until an unexpanded node is found. Evaluations are then updated along the path back to the root.

2.2 ESPRIT

2.2.1 The formulas

We first convert the v_m's and n_m's to a policy $q = (q_1, \ldots, q_M)$, and then to the final policy $r = (r_1, \ldots, r_M)$ which combines q with the static policy p. We set $r = p$ if all n_m are zero, and assume at least one is nonzero from now on.

Suppose the v_m's are sorted so that $v_1 \geq v_2 \geq \cdots \geq v_M$. (This is not necessary for the algorithm but simplifies the notation.) First we set

$$z_m = {n_m}^{\alpha} \tag{3}$$

where α is a parameter. We set Z to be the mean of these over moves:

$$Z = \frac{\sum_j z_m}{M}. \tag{4}$$

Now we can define the key function $f()$ as

$$f(x; \gamma, \sigma) = \left(1 + \frac{x}{\sigma Z^{-1/2}}\right)^{-\gamma} \tag{5}$$

where γ and σ are parameters. Policy q is now obtained by

$$q'_m = f(v_1 - v_m; \gamma, \sigma) \text{ and } q_m = \frac{q'_m}{\sum_j q'_j} \tag{6}$$

Finally the transitional probabilities $r_1, \ldots, r_M$ are then obtained by

$$r'_m = \frac{\lambda p_m + z_m q_m}{\lambda + z_m} \text{ and } r_m = \frac{r'_m}{\sum_j r'_j} \tag{7}$$

where λ is a parameter. We explain how the r_m are used to search the tree in section 2.3.

2.2.2 Motivation

We provide some explanation and justification for the choices made here. Each v_m is the mean of n_m values, and if they were independent and identically distributed random variables, the effective sample size (ESS) would be n_m, and the variance of v_m would scale with $1/n_m$. The evaluations in a game tree are certainly not independent, and the ESS will be less than n_m. Equation (3) captures this idea (with $\alpha < 1$). We do not actually want the ESSs for the individual v_m's but rather for functions which involve all of them. The way that the z_m are used will be discussed after explaining other aspects.

Consider a node with a particular move $b \geq 3$, with n_1, n_2, n_b all large and suppose that neither $(v_1 - v_b)$ and $(v_2 - v_b)$ are close to zero. Nonetheless b may be the best move. This is the sort of situation that search algorithms struggle with: usually evaluations based on much lookahead are accurate, but occasionally big errors are made. It is extremely unlikely that both v_1 and v_2 are severe overestimates, so the probability that b is the best move is approximately the probability that the noise distribution for v_b generates a value smaller than $-(v_1 - v_b)$. This makes the tails of the noise distributions very important, and suggests that for nodes with many visits, a power law curve like $f(x; \gamma, \sigma)$ is more suitable than an exponential which leads to a softmax. We use $(v_1 - v_m)$ to preserve the location independence of the softmax.

Note that for unexpanded moves (when $n_m = 0$ and $v_m = 0$), equation (7) sets $r'_m = p_m$ since $z_m = 0$ for unexpanded moves. However, equation (6) sets $q'_m = f(v_1; \gamma, \sigma)$, so that the unexpanded moves do have some influence on the transitional probabilities for expanded moves. For example, increasing the number of unexpanded moves reduces the influence of expanded moves on r, other things being equal.

$Z^{1/2}$ plays a role analogous to the 'learning rate' in Boltzman exploration. The statistical interpretation is that Z approximates the ESS per move, so that $1/Z$ is expected to be proportional to the variance of the noise in the v_m's, so that $Z^{-1/2}$ is a suitable scaling factor for $f()$. I have no reason to suppose that equation (4) has the right form. In the situation involving moves 1, 2, and b described above, setting Z to the harmonic mean of z_1, z_2, and z_b rather than the mean seems appropriate, but a harmonic mean makes no sense if there are any unexpanded moves. An exact analysis of the general situation appears difficult, involving summing over all permutations of $\{1, 2, \ldots, M\}$.

Some examples may clarify the behavior of ESPRIT when Z is small. If $n_1 = 1$ with the rest 0, then $Z^{-1/2} = \sqrt{M}$, so the ratio of q_1 to q_m for $m > 1$ is $1 : f(v_1/(\sigma\sqrt{M}))$ representing large uncertainty in this case. Suppose $n_1 = n_2 = 1$ with the rest 0, and that v_1 and v_2 are noisy estimates of true evaluations t_1 and t_2. Especially when v_1 and v_2 are close, there is uncertainty about whether $t_1 > t_2$ or $t_2 > t_1$, and hence whether q_1 or q_2 should be biggest. Thus q_1 and q_2 should be close in this case and $Z^{-1/2} = \sqrt{M/2}$ achieves this.

Equation (7) can be viewed as an approximate Bayesian update of the initial policy vector using the policy based on the number of 'effective observations' z_m of the latter. The parameter λ can be viewed as a count of prior observations in the static policy. In this equation, q_m depends more on on v_m than other v_j most of the time, but z_m is an approximation which might be improved upon.

2.2.3 Choosing parameter values

We have defined four parameters, namely: α for effective sample size; γ which determines the shape of the curve for converting v_m's to q_m's; σ, the scale for this curve; and λ, the weight given to the p_m's versus the q_m's when combining them. We now consider how to choose their values.

The optimal form of the function which provides the ESS of an evaluation based on n_m static evaluations depends on the algorithm itself, so the best choice probably requires a very deep analysis. It appears that $z_m = \sqrt{n_m}$ is about right for ESPRIT on the simulated data used here, so $\alpha = 0.5$ is a sensible choice.

The parameters γ and σ are important. If large values are used for both γ and σ, formula (5) becomes close to exponential. As argued above, this is not likely to be optimal. The parameter σ determines the sharpness of the policy derived from evaluations. Large σ makes broad shallow trees; small σ makes narrow deep trees. It appears that $\gamma = 3, \sigma = 0.07$ is a good choice for the data used here.

The optimal value of λ depends on the relative accuracy of the static policy and the static evaluations. It appears that $\lambda = 1$ is reasonable for the data here.

2.3 Searching using realization probabilities

For the results in section 4, we use a very similar algorithm to PUCT. The next node to be expanded is found iteratively from the root, choosing the move at each node which maximizes a score until an unexpanded node is found. The only difference is that $\text{argmax}_j(v_j + u_j)$ (from equation 1) is replaced by $\text{argmax}_j(r_j/(n_j + 1))$. The idea is that moves with large $r_m/(n_m+1)$ are under-represented. To make a final move choice at the root, either $\text{argmax}_j(r_j)$ or $\text{argmax}_j(n_j)$ can be used. We use the latter here, the same as PUCT.

Realization probabilities are suited to parallel tree search. The transitional probabilities give a more principled way of allocating threads than a score like the PUCT formula of equation (1) (but see the next section 2.4). As mentioned in the introduction, the iterative stochastic method of Igarashi et al. (2019) can be used to choose a node to expand, and this method can be parallelized.

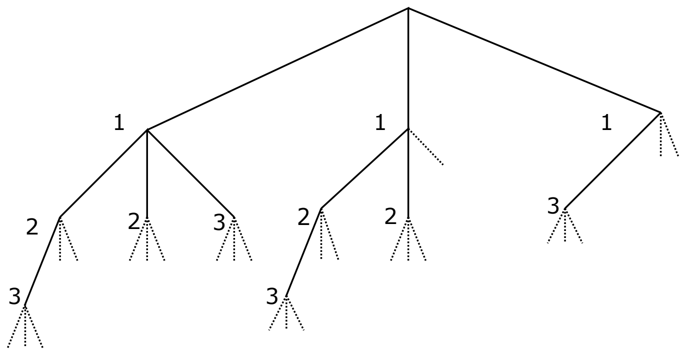

Figure 1: Illustration of a SIMD algorithm for ESPRIT, with 4 threads and a tree with branching factor 3, after three waves. The fourth wave will chose among the unexpanded moves, which are shown as dotted lines.

We briefly describe an algorithm suitable for SIMD architectures. The idea is to expand the tree in a series of 'waves' (see Figure 1) using B threads. Each wave contains up to B nodes, each of which is one ply deeper than some node in some previous wave. Thus wave w contains nodes of depths no more than w. To construct the next wave, B threads are assigned to the root, and the wave is advanced one depth at a time in a series of steps. Within one step a node x can be assigned some range $\{i : b_0 \le i < b_1\}$ of threads. The transitional probabilities are summed over expanded and unexpanded moves, and a decision is made about how many threads to assign to unexpanded moves from x. Then the remainder are allocated to child nodes in approximate proportion to their transitional probabilities for the next step. Eventually a thread will either be assigned to an unexpanded move, or will idle because it reached a node with no children and fewer than $b_1 - b_0$ moves. Evaluations can be backed up to the root in parallel too.

2.4 Reinterpreting PUCT

PUCT is described as improving the evaluations given a policy. I have described ESPRIT as improving the policy given evaluations. I do not think this distinction is of much importance, since both tasks must be solved together. Since C and n_s do not vary with m,

$$\left(p_m + \left(\frac{1+n_m}{n_s}\right)\frac{\sqrt{n_s}}{C}v_m\right)\Big/(1+n_m) \tag{8}$$

has the same order properties as formula (1) and will result in the same searches. This can be be rewritten as

$$x_m = p_m + \left(\frac{\sqrt{n_s}}{C}v_m\right)p'_m \tag{9}$$

where $p'_m = (1+n_m)/n_s$ and moves are selected using $\text{argmax}_j(x_j/(1+n_j))$. Now p' is almost a policy based on visit counts (it requires multiplication by $n_s/(n_s+M-1)$ to normalize it), so PUCT can be reinterpreted as a kind of policy refinement. Compare equation (7) for the transitional probabilities in ESPRIT. This reinterpretation could be used to allocate threads in a parallelized version of PUCT, as outlined in scetion 2.3.

The algorithms outlined in section 2.3 can therefore be applied to PUCT.

In Grill et al. (2020), another interpretation of PUCT, as regularized policy optimization, is described. They derive an updated policy using

$$\hat{y} = \text{argmax}_{y \in S} \sum_j v_j y_j - R(y, p) \tag{10}$$

where S is the probability simplex $\{y \in \mathbb{R}^M : y_j \geq 0, \sum_j y_j = 1\}$, and $R(y, p)$ is a regularization term, which prevents y from becoming 'too far' from p, and is weighted so that its effect diminishes as the number of evaluations increases. The optimal value $\hat{y}$ is not used to provide realization probabilities, but instead is used in a formula which replaces u_m in equation (1). For a particular choice of $R()$, PUCT can be reinterpreted as regularized policy optimization.

3 Simulating game trees

3.1 Outline

The simulation aims to mimic the properties of typical chess game trees, though it is flexible enough to mimic a wide variety of game trees. The method uses a pseudo-random number generator (PRNG) which is seeded for each node. A node is identified by the sequence of move indices from the root to the node, plus an index for the tree, and this sequence is used for seeding. Evaluations are from the point of view of the player to move and are in $[0, 1]$. There is no end point to the game; in a game between unequal players, the evaluations will tend to 1 for the strong player, but (given exact computation) will never reach it.

For each node, the number of moves M is sampled, then 'true' evaluations $t_1, \ldots, t_M$ are generated. A 'true' evaluation is not the final game score given perfect play but the expected score in a game between two players. Noise $x_1, \ldots, x_M$ is generated, and the 'estimated' evaluation v_m for move m combines t_m, x_m, and the noise inherited from the parent. For the estimated policy, another, independent noise vector $y_1, \ldots, y_M$ is generated. A value w_m for move m is found from t_m, x_m, y_m, in a similar manner to v_m and correlated to v_m via x_m. The w_m are then softmaxed to produce a policy. The estimated evaluations for moves are correlated with the parent's estimated evaluation, via the inherited noise. The estimated policy is correlated with the estimated evaluations via the x_m.

3.2 Details

Number of moves. A normal distribution is used to generate M. The mean of the normal distribution is $(M_{mean} + M_P)/2$ where M_P is the number of moves of the parent, and the standard deviation is M_{sd}. M is then the nearest integer in the range 1,2,...M_{max} to the sampled normal. For the root node, $M_P = M_{root}$.

The true evaluations. $u_1, \ldots, u_M$ are sampled from an exponential with rate $\sqrt{M}/T_{scale}$. Then

$$t_m = \frac{w}{w + (1 - w) \exp\left(-\sum_{j=1}^{m} u_j\right)} \tag{11}$$

where w is 1 - (parent's true evaluation). This produces $t_1 < \cdots < t_M \in [0, 1]$. Thus the player at the node will try to choose move 1, which corresponds to the smallest t_m (the worst for the opponent who will move next).

The estimated evaluations. $x_1, \ldots, x_M$ are sampled from a Student's t-distribution with E_{df} degrees of freedom and scaled by E_{ch}. Then

$$x'_m = \frac{x_m - E_{par} x^*}{\sqrt{1 + E_{par}^2}} \tag{12}$$

where x^* is the noise inherited from the parent node. The estimated evaluation for move m is then

$$v_m = \frac{t_m}{t_m + (1 - t_m) \exp(x'_m)} \tag{13}$$

The minus sign in the numerator of (12) is there because x^* is from the parent's point of view. A positive value for the parameter E_{par} corresponds to positive correlation between evaluations from the point of view of a single player. This could happen if an evaluation function misses something about a board position and continues to miss it after a move. A negative correlation could happen during a capture sequence in chess. The division by $\sqrt{1 + E_{par}^2}$ keeps the variance of the noise similar throughout the tree. The shape of the noise does change a little as one goes deeper into the tree.

The estimated policy. $y_1, \ldots, y_M$ are sampled from a Student's t-distribution with P_{df} degrees of freedom and multiplied by P_{ch}. Then $y'_m = y_m + P_{eval} x_m$,

$$y''_m = \frac{t_m}{t_m + (1.0 - t_m)\exp(y'_m)}$$

and finally, the estimated policy value for move m is

$$p_m = \frac{\exp(-P_{sm} y''_m)}{\sum_j \exp(-P_{sm} y''_j)} \tag{14}$$

Table 1 summarizes the parameters for the simulated trees.

Table 1: Simulation parameters

Parameter	Description
	Moves
M_{root}	Number of moves of parent of root
M_{mean}	Mean number of moves
M_{sd}	Standard deviation of number of moves
M_{max}	Maximum number of move
	True evaluations
T_{root}	True evaluation at root
T_{scale}	For spacing of true evaluations
	Static evaluations
E_{df}	Shape of distribution of noise
E_{par}	Amount of noise inherited from parent
E_{ch}	Amount of per-child noise
	Static policy
P_{df}	Shape of distribution of noise
P_{ch}	Amount of per-child noise
P_{eval}	Noise shared with static evaluations
P_{sm}	Rate for softmax

4 Results

4.1 The simulated trees

For the results presented here, game trees were generated using the parameters in Table 2. For each tree, a uniform random choice is made from the list of values following each parameter. In Test A, these are intended to mimic the game trees, evaluations, and policies which might come from a small neural net applied to chess. In Test B, the values are intended to be like a higher quality evaluation in go: the number of moves is increased, and the amount of noise (E_{ch}, P_{ch}) is reduced. Test C is the same as Test B, but with exactly 2 moves from each state. These are not intended to constitute a systematic study, but only to illustrate some interesting points.

Figure 2 gives an indication of the accuracy of the evaluations and policies in Test A. These were generated by sampling a large number of static evaluations for the moves from the root node, and static policies for the root node, using the simulation scheme of section 3, and counting the number of times the ith biggest evaluation or transitional probability corresponded the optimal move. Note that the policy is somewhat more informative than the evaluations in terms of ordering the moves. Samples of estimated policies were also used to choose a reasonable value for P_{sm}, by comparing Pr(best|rank 1)/Pr(best|rank M) with the geometric mean of $\max_j(p_j)/\min_j(p_j)$. It was found that, for the tests conducted here, $P_{sm} = 10$ produced policies which were not much too sharp or much too flat by this measure.

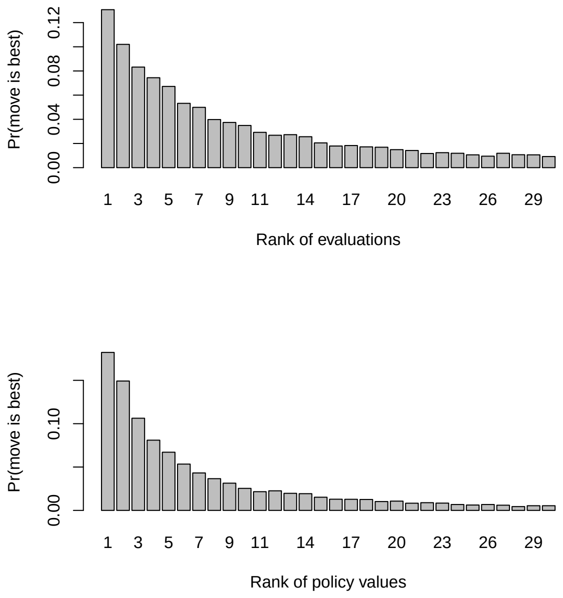

Figure 2: These are based on simulations for Test A with $M = 30$ moves. The top histogram is based on samples of static evaluations, one per move. The lower histogram is based on static policies. A value with rank i is the ith biggest among the moves. See text for more details.

4.2 Algorithm parameters

For ESPRIT, the settings $\alpha = 0.5, \lambda = 1.0, \gamma = 3, \sigma = 0.07$ were used for Tests A, B, and C. These were chosen after experimenting with various variations in the formulas used, and parameter values, using trees similar to, or identical to, those of Test A. Thus ESPRIT has been well-tuned for this data. No further tuning was done for Tests B and C, however.

For PUCT, we left $c_1 = 196524$, and experimented with different values for c_0. We tested 19 values close to the powers of $2^{1/3}$ ranging from 0.25 to 16 on trees of size up to $2^{16} = 65536$, to locate a good value for c_0. The performance was not very sensitive to the value. For Test A, we chose 2.5, for Test B, we chose 6.4, and for Test C, we chose 0.4.

4.3 Test A

Figure 3 shows the performance as the number of playouts ranges from 1 to $2^{20} \approx$ one million. For the larger tree sizes, PUCT has mean regret around double that of ESPRIT, corresponding to needing four times the tree size to achieve a similar result. From 2^{10} to 2^{18} both show a straight line with similar slope on the log-log plot, with mean

Table 2: Game tree simulation parameter values

Parameter	Test A	Test B	Test C
Moves			
M_{root}	10 30 50	200	2
M_{mean}	10 30 50	200	2
M_{sd}	5.0	5.0	0.0
M_{max}	100	300	2
True evaluations			
T_{root}	0.1 0.3 0.5 0.7 0.9	Same as A	Same as A
T_{scale}	0.1 0.2 0.3 0.4	0.1 0.2 0.3	Same as B
Static evaluations			
E_{df}	2.0 3.0 4.0	Same as A	Same as A
E_{par}	-1.0 0.0 1.0 5.0	Same as A	Same as A
E_{ch}	0.2 0.3 0.4 0.5	0.2 0.3	Same as B
Static policy			
P_{df}	2.0 3.0 4.0	Same as A	Same as A
P_{ch}	0.1 0.2 0.3	0.1 0.2	Same as B
P_{eval}	0.1 0.2 0.3	0.1	Same as B
P_{sm}	10.0	Same as A	Same as A

regret approximately proportional to (tree size)$^{-0.33}$. PUCT deviates from this for largest two sizes, and ESPRIT appears to do the same for the largest size. The percentage of correctly chosen moves increases from 21% for one playout to 76% (ESPRIT) and 69% (PUCT) for 2^{20} playouts.

Results on subsets of trees with particular simulation parameter values are shown in Table 3. Both methods do worse with more moves (M_{mean}) though PUCT gets worse faster. Both do worse with the true evaluation $T_{root} = 0.5$, presumably because evaluations tend to get squashed near 0 and 1, so the regret is smaller. Neither method seems sensitive to T_{scale}.

Both methods unsurprisingly do worse with smaller E_{df} and larger E_{ch}, both of which make the evaluations noisier. PUCT is relatively poor when correlation between parent and child evaluations are 0 or negative ($E_{par} \leq 0$) and relatively good with high positive correlation between parent and child evaluations ($E_{par} = 5$).

Turning to the policy, the trends are in the expected direction for P_{df} and P_{eval}, with more dramatic changes in PUCT than ESPRIT. Both methods are surprisingly insensitive to the amount of uncorrelated noise P_{ch} in the policy.

4.4 Test B

Figure 4 shows the performance of ESPRIT and PUCT on Test B, where the number of moves from a state is around 200. ESPRIT becomes slightly worse between 64 playouts and 256 playouts. This is not understood, though remember that ESPRIT's parameters have not been adjusted for this test. ESPRIT substantially outperforms PUCT on this data, with roughly half the regret, although the large scale trends are less clear than with Test A. The percentage of correctly chosen moves increases from 8% for one playout to 50% (ESPRIT) and 41% (PUCT) for 2^{20} playouts.

4.5 Test C

Figure 5 shows the performance of ESPRIT and PUCT on Test C, where the number of moves from a state is always 2. PUCT has mean regret around double that of ESPRIT until 2^{12}=4096, and then becomes more than double. PUCT deviates from this from 2^{14}=16384, and ESPRIT deviates from 2^{18}=262144. The percentage of correctly chosen moves increases from 68% for one playout to 95.7% (ESPRIT) and 92.5% (PUCT) for 2^{20} playouts.

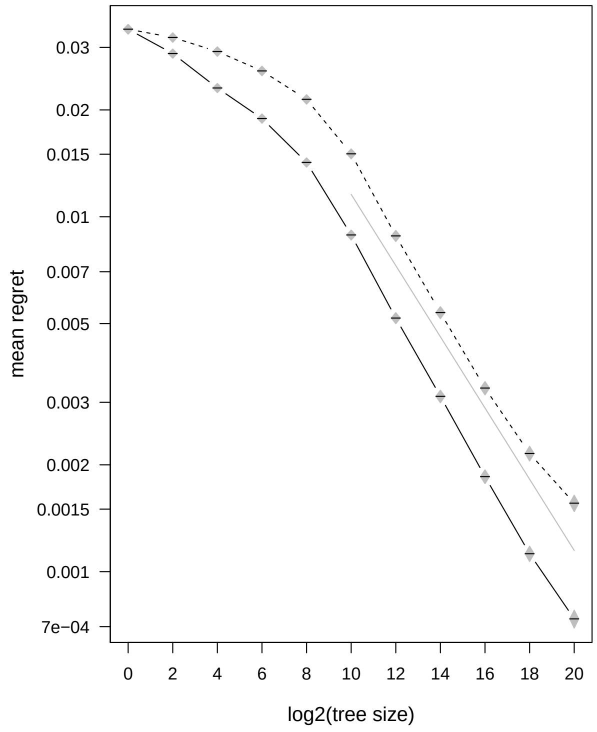

Figure 3: The performance of ESPRIT (solid line) and PUCT (dashed line) for Test A. Mean regret over 10000 trees is plotted against tree size (number of playouts) on a log-log scale. The diamonds extend to ± 2 standard deviations. The grey line extends from tree size 2^{10} to 2^{20}, and is proportional to (tree size)$^{-1/3}$.

Table 3: Subsets of trees

Parameter	Value	Set size	ESPRIT	PUCT	Difference
M_{mean}	10	3271	1.66	2.34	0.68
M_{mean}	30	3408	1.89	3.22	1.33
M_{mean}	50	3321	2	3.9	1.9
T_{root}	0.1	2030	1.36	2.39	1.02
T_{root}	0.5	2043	2.37	3.61	1.24
T_{root}	0.9	1966	1.29	2.81	1.52
T_{scale}	0.1	2521	1.76	2.78	1.02
T_{scale}	0.2	2549	1.89	3.23	1.34
T_{scale}	0.3	2474	1.93	3.49	1.56
T_{scale}	0.4	2456	1.83	3.15	1.32
E_{df}	2	3331	2.36	3.72	1.36
E_{df}	3	3386	1.69	2.9	1.21
E_{df}	4	3283	1.5	2.86	1.36
E_{par}	-1	2498	1.57	3.2	1.63
E_{par}	0	2507	1.4	2.89	1.49
E_{par}	1	2445	2.24	3.47	1.23
E_{par}	5	2550	2.2	3.09	0.89
E_{ch}	0.2	2542	1.31	2.69	1.38
E_{ch}	0.3	2486	1.65	3.06	1.41
E_{ch}	0.4	2495	2.07	3.15	1.09
E_{ch}	0.5	2477	2.4	3.75	1.35
P_{df}	2	3342	2.11	4.65	2.54
P_{df}	3	3351	1.76	2.61	0.85
P_{df}	4	3307	1.68	2.21	0.53
P_{eval}	0.1	3347	1.63	2.08	0.45
P_{eval}	0.2	3377	1.88	2.96	1.08
P_{eval}	0.3	3276	2.05	4.47	2.42
P_{ch}	0.1	3238	1.69	3.09	1.4
P_{ch}	0.2	3438	1.89	3.22	1.33
P_{ch}	0.3	3324	1.97	3.17	1.2

Notes: This table shows the behavior of ESPRIT and PUCT on various subsets of 10000 trees each of size $2^{16} = 65536$, corresponding to particular simulation parameter values (first two columns). The third column shows the number of trees in the subset. The mean regrets, multiplied by 1000 for easier reading, for ESPRIT and PUCT are in the fourth and fifth column, and the difference in the sixth column.

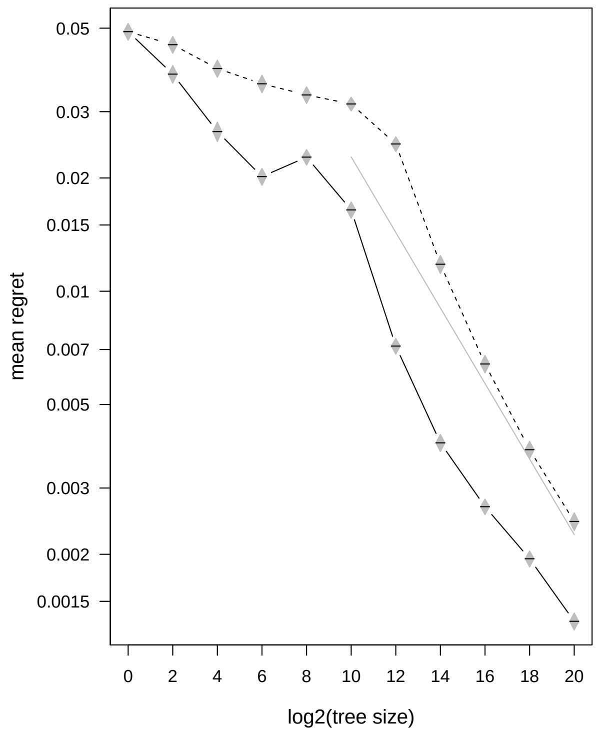

Figure 4: The performance of ESPRIT (solid line) and PUCT (dashed line) for Test B. Mean regret over 4000 trees is plotted against tree size (number of playouts) on a log-log scale. Other details as Figure 3.

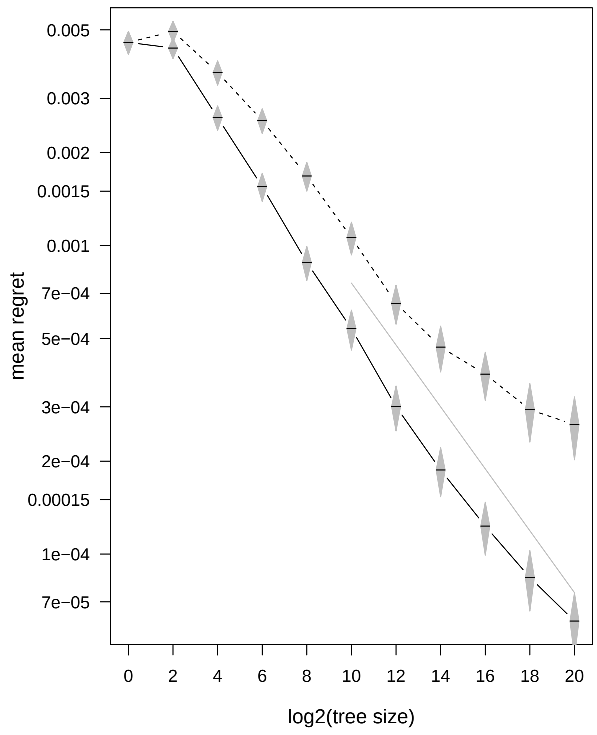

Figure 5: The performance of ESPRIT (solid line) and PUCT (dashed line) for Test C. Mean regret over 4000 trees is plotted against tree size (number of playouts) on a log-log scale. Other details as Figure 3.

5 Discussion

5.1 Simulations

There seems to be little work using simulated data for game tree searches. Instead algorithms are compared by playing large numbers of particular games using particular evaluation functions. It takes a huge computational effort to be able to draw general conclusions from these tests, and those conclusions may say little more than 'Algorithm X is usually superior to algorithm Y'. If ESPRIT proves disappointing when used in actual games, it will indicate that there is something unrealistic about the nature of the tree simulations used here. By examining cases where it fails, it may be possible to make the simulated data more realistic, and develop appropriate search algorithms for the new data. Progress in other areas of science (such as phylogenetics with which I am familiar) has been made in this way. Simulated data is not just a convenience, it is a model of some aspect of reality, and one can learn from its deficiencies.

5.2 Implementations

Practical implementation of a high quality game engine using ESPRIT using a single search tree will require accurate evaluations, otherwise the memory required for the tree will become prohibitive. ESPRIT could be used instead of PUCT in AlphaZero's reinforcement algorithm, but the computational resources required to test this are huge.

Another option is use a main tree and short-lived subtrees at large depths. The SIMD implementation of ESPRIT outlined in section 2.3 opens the possibility of moving some of the search into a GPU. The main search tree would be stored in the host device, and when it chose a node to expand, a GPU routine would perform a small search of say 1000 playouts below this node. From the host's point of view, it would be as if a single high quality evaluation was obtained. Within the GPU, a group of threads would identify a batch of nodes to evaluate, and the evaluations would be parallelized over board positions. A small fully connected neural net may be more attractive than a convolutional one in this situation. The memory bandwidth for weights is not the problem it usually is, because each weight is used on many board positions.

5.3 Theory

There are also possibilities for progress of a more theoretical nature, which we briefly enumerate.

1. The optimal c_0 values we found for Test A and Test B were substantially bigger than those used in AlphaZero, and varied with the number of moves. The optimal values for the data here are around $\sqrt{M}/2$. It would be interesting to know why this is the case. Perhaps there is something about the simulated trees used here which is different from game trees in chess, shogi, or go. Perhaps the reinforcement learning context of AlphaZero has this effect on the optimal value of c_0.
2. The mathematical connections between ESPRIT and the work of Grill et al. (2020) could be explored.
3. There should be a connection between the formula used for ESSs in equation (3) and the curves for mean regret in the results. On the limited tests done so far, it appears that $z_m = {n_m}^{1/2}$ is about right for ESSs whereas the results suggest that (mean regret) $\propto$ (tree size)$^{-1/3}$ is about the best that can be achieved with ESPRIT.
4. The discussion in section 2.2.2 suggests that the way the z_m's are used (formulas 4 and 7) could be improved.
5. A careful analysis of the noise distribution of particular static evaluations and static policies should improve performance on moderately sized trees. Eventually one might automate the whole process, and tune the search algorithm simultaneously with training a neural net.

Software

A snapshot of the C++ code used in this paper is available from `http://indriid.com/workingnotes2021.html`.

References

Tuan Dam, Pascal Klink, Carlo D'Eramo, Jan Peters, and Joni Pajarinen. Generalized mean estimation in Monte-Carlo tree search. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 2397–2404. International Joint Conferences on Artificial Intelligence Organization, 7 2020. doi: 10.24963/ijcai.2020/332. URL `https://doi.org/10.24963/ijcai.2020/332`. Main track.

Ahmed A. Elnaggar, Mostafa Abdel Aziem, Mahmoud Gadallah, and Hesham El-Deeb. A comparative study of game tree searching methods. *International Journal of Advanced Computer Science and Applications*, (5), 2014.

Jean-Bastien Grill, Florent Altché, Yunhao Tang, Thomas Hubert, Michal Valko, Ioannis Antonoglou, and Remi Munos. Monte-Carlo tree search as regularized policy optimization. In Hal Daum III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 3769–3778. PMLR, 13–18 Jul 2020. URL `http://proceedings.mlr.press/v119/grill20a.html`.

Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Tobias Pfaff, Theophane Weber, Lars Buesing, and Peter W. Battaglia. Combining q-learning and search with amortized value estimates, 2020.

Harukazu Igarashi, Yuichi Morioka, and Kazumasa Yamamoto. Learning position evaluation functions used in Monte Carlo softmax search. *Unknown Journal*, January 2019. URL `arXiv:1901.10706`. Publisher Copyright: Copyright © 2019, The Authors. All rights reserved. Copyright: Copyright 2020 Elsevier B.V., All rights reserved.

T. Kimura, T. Ugajin, and Y. Kotani. Bigram realization probability for game tree search. In *2011 International Conference on Technologies and Applications of Artificial Intelligence*, pages 260–265, 2011. doi: 10.1109/TAAI.2011.53.

A Kirii, Y Y Hara, H Igarashi, Y Morioka, and K Yamamoto. Stochastic selective search applied to shogi. In *Proceedings of the 22th Game Programming Workshop (GPW2017)*, pages 26–23, 2017. (in Japanese).

Kenneth L. Lange, Roderick J. A. Little, and Jeremy M. G. Taylor. Robust statistical modeling using the t distribution. *Journal of the American Statistical Association*, 84(408):881–896, 1989. doi: 10.1080/01621459.1989.10478852. URL `https://doi.org/10.1080/01621459.1989.10478852`.

Brian D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, 1996. doi: 10.1017/CBO9780511812651.

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. ISSN 0036-8075. doi: 10.1126/science.aar6404. URL `https://science.sciencemag.org/content/362/6419/1140`.

Yoshimasa Tsuruoka, Daisaku Yokoyama, and Takashi Chikayama. Game-tree search algorithm based on realization probability. *ICGA Journal*, 25(3):145–152, 2002. doi: 10.3233/ICG-2002-25304.

Mark H. M. Winands and Yngvi Björnsson. Enhanced realization probability search. *New Mathematics and Natural Computation*, 04(03):329–342, 2008. doi: 10.1142/S1793005708001070. URL `https://doi.org/10.1142/S1793005708001070`.