Monte carlo methods for estimating game tree size

Daniel S. Abdi
dshawul@yahoo.com

April 25, 2013

Abstract

Partial game tree sizes can be estimated using different methods among which the most accurate is conducting monte-carlo simulations of random games. This process resembles monte-carlo tree search (MCTS) method as used for game playing in Computer Go, but it has also some interesting differences. For instance the reward assigned to a simulation is actually contained in the path taken, not in the terminal position as is common in game playing. The article investigates a new application of MCTS that is not meant for game playing but estimating game tree size itself. The simplicity of the problem makes it an ideal playground for studying MCTS method in general. All the components of MCTS namely simulation, back-propagation, selection and expansion are applicable for this problem as well, and comparisons with MCTS for game playing are made whenever possible. The selection policy most commonly used in MCTS, i.e. upper confidence bound (UCB) formula, is inappropriate for this application thus new formulae are derived for optimal tree and default selection policies. The result is a partial game tree size estimator that converges rapidly to a lower variance.

1 History of perft calculations

Game tree size is defined as the total number of games that can be played from the initial position. This number is also equal to the number of leaf nodes in a game tree. The partial game tree size computed from any position and up to a certain depth d is informally known as perft(d). In the area of computer chess programming, perft is used to verify correct implementation of move generation and to some extent for comparing efficiency. Perft is also used for similar purpose in other games such as checkers, but from here on we assume perft to imply calculations on game of chess.

The first computation of perft were done with a Cobol program written by Smith [1978] as reported in a magazine article.

> The program automatically generates the information that after the first three half moves - white black white - there are 8902 possible three-move games.

Ken Thompson may have calculated perft(3) and perft(4) earlier than this date with his dedicated chess computer Belle. Edwards [2012] was the first to compute perft(5) through perft(9), and has since been actively involved in perft computations. Exact perft numbers have been computed and verified up to a depth of 13 and are now available in the online integer sequence database Edwards [2012]. Also an unverified claim for perft(14) is given by Osterlund [2013]. The method used for this latest claim calculates and stores all the unique nodes at ply=11 to speed up the computation significantly. Similar optimization is also used by Bean [2003] to compute perft(10) and also by Paul Byrne to compute perft(12) and perft(13). A disadvantage to this otherwise very good optimization is that a breakdown of perft numbers by move is not available. Thus the reported number needs verification by methods that are able to provide perft breakdown up to a certain depth, similar to that used by Edwards [2012] to compute perft(13). Perft(14) using this method will probably take few years to compute but faster completion is not to be ruled out. Early computation of perft(11) by Bertilson [2011] was done using a distributed project. However the latest largest perft computed so far i.e. perft(14) is done on few computers running for a couple of months, which is a testament to hardware and algorithmic improvements.

2 Perft estimation

This article focuses on methods used for estimating perft figures, and not of computing the exact values as discussed in the previous section. The motivation for this work came from an informal competition to estimate perft(13) before the actual number was revealed by Edwards [2011]. Among the different methods that came into light during the competition, the monte-carlo methods were clear winners. Clearly these methods have been used in the past for computing perft estimates. Labelle [2007] used what he called ”random pruning” method to compute estimates of perft(13) through perft(20). The details of his method are not available, but the basic idea seems to be of monte-carlo estimation. Further investigation as to the first use of monte-carlo methods for estimating perft has not been done, but we suspect it maybe quite old since it is a simple application of original monte-carlo method proposed in the 1940s [Eckhardt, 1987]. However estimating perft by other means is certainly not a recent effort. Anthony [1878] mentions an estimate for perft(8) which is now proven to be off by 275%.

2.1 Branching factor estimation

The most straight forward way of estimating perft(d) numbers uses product of estimated branching factors at each ply. This is the method used by Anthony to compute a crude approximation of perft(8) as shown in 1. Similarly an estimate of perft(20)=1.69e29 is found using a suggestion he made to use a constant branching factor of 30 for the rest of the plies.

$$perft(8) = 20 * 20 * 28 * 29 * 30 * 31 * 33 * 32 = 318,979,584,000 \pm 20\% \quad (1)$$

Labelle noted that the estimate could have been better if he used a close to exact estimate of perft(4) available at the time. Using the now known exact value of perft(8) and same branching factor for the rest of the plies, perft(20) is estimated to be 4.517e28. Similarly using exact value of perft(14), perft(20) is estimated to be 4.511e28 which is surprizingly not very far from the previous estimate. Getting exact estimates of branching factors for six more additional plies did not help much. Probably the best estimate for perft(20) comes from Labelle that used monte-carlo method to estimate it at 8.35e28. Assuming this estimate to be exact, we can conclude that a severe error in estimating the branching factor in only one of the plies is enough to give a poor estimate. Computing estimate of perft(15) will not be so easy even though perft(14) is now known and the task is to reduced to estimating the average branching factor for the last ply. However this may be a potential improvement to speed up monte-carlo perft estimations by skipping the first 14 plies and collecting data only from the last ply.

The relative error in this method of perft estimation is a cumulative of the errors in the estimates of branching factors at each ply. Hence given a relative error of $\Delta x/x = 1\%$ for estimates of branching factors, the relative error in estimating perft(20) will be approximately 20%. This is a result of uncertainty propagation in a product of variables as given in equation 2. It is to be noted that if the errors are independent and random,as it may well be the case with monte-carlo simulations, the cumulative error can be much less than 20%.

$$\begin{aligned} P &= \textstyle\prod_{i=0}^{d} BF_i \\ \frac{\Delta P}{P} &\sim \textstyle\sum \frac{\Delta BF_i}{BF_i} \end{aligned} \quad (2)$$

2.1.1 Odd-Even interpolation

It is interesting to look at the variation of perft(d) from ply to ply using so far computed values of exact perft up to ply=14, and estimates found from monte-carlo simulations for later plies as shown in figure 1. The ratio perft(d)/perft(d-1), which is the average branching factor at the last ply=d, is plotted against depth. An interesting zig-zag pattern is observed with the branching factor increasing when going from even to odd plies, and decreasing from odd to even plies. Labelle observed that this pattern continues beyond 14 plies with perft estimates computed using monte-carlo simulations. Perft estimation using branching factors can be significantly improved by curve fitting to this observed behavior. An example spline interpolations separately for odd and even plies are shown in figure 1. For this particular case the even plies are better predicted than the odd ones. During the perft betting competition better interpolation functions were used for a much better result. Some of the functions used include: linear, cubic and high order polynomials, inverse interpolation etc.

It is not at all strange that the branching factor increases with depth on average due to increase in mobility of pieces, and that it must eventually come to stagnation. However the odd zig-zag pattern is not straightforward to explain. The pattern has a peculiar similarity to the well known "odd-even effect" of alpha-beta tree search, where the branching factor at even plies is significantly smaller than those at odd plies. Having said that, the inherent branching factor fluctuation observed in perft is unrelated to this phenomenon, since the latter is an effect introduced by alpha-beta pruning and not an inherent characteristic of Minmax tree. A possible explanation is that the opponent tends to reduce our mobility on average, which is specially true at the initial position where all the pieces are parked on back ranks for best mobility. This is not to be confused with an opponent that tries to minimize our mobility, as is the case with Minmax search. Making a random move will reduce our mobility on average due to the nature of the game in the opening positions. This may not be true in other games or even for perft computed from different start position.

2.2 Middle ground method

Another way of estimating branching factors is taking random samples of positions at a given ply and then computing the average perft(1) counts. This clearly overlaps with the monte-carlo methods that will be discussed in detail later, with the main difference being mean perft(d) for $d > 1$ is

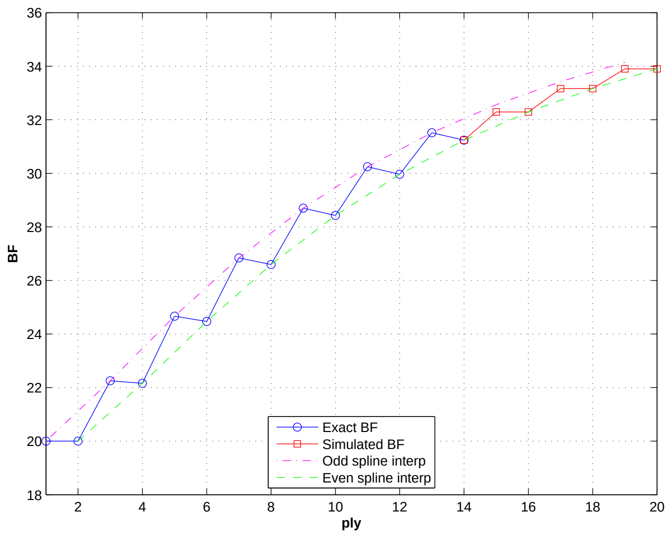

Figure 1: Branching factor interpolation

directly computed instead of mean branching factors. A variation of this method was used by Adam Hair during perft betting competition to compute the branching factor at ply=12 from available chess game databases. Clearly the subset of often played positions is much smaller than the total perft(12), hence getting an accurate estimate this way is problematic. Most importantly the estimate may be biased towards most frequently played positions. While this is in no way a confirmation of this observation, indeed the average perft(1) computed this way gave a much larger value of 34.75 compared to the exact value of 31.5 ,using a sample of 37640 unique ply=12 positions. It is to be noted that estimates of branching factors at shallower plies are more accurate for a given number of simulations. This is merely due to finite population size even though the population variance is assumed to be same at all plies. After n simulations in a population of N=perft(d) variance will be reduced to the value given in equation 3.

$$V_{\overline{x}} = \frac{\sigma^2}{n}(\frac{N-n}{N-1}) \tag{3}$$

A more severe problem with conducting monte-carlo simulations to first determine average branching factors and then the perft estimate is that the number of moves available to a side on two consecutive plies are correlated. This is clearly observed in figure 2, where the perft values are highly underestimated with the method that computes mean branching factors. The

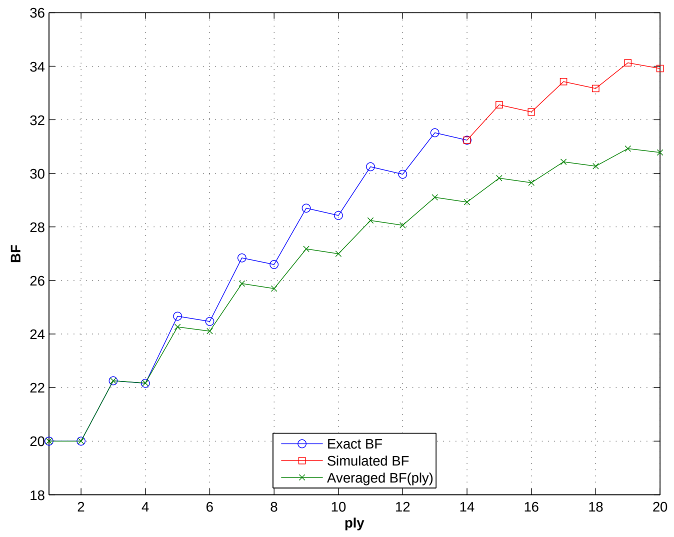

Figure 2: Averaging branching factors

covariance term in equation 4 must be positive for this to be true, confirming the fact that there is a positive correlation between branching factors. This problem effectively renders the method unusable for large depth perft predictions.

$$E(XY) = E(X) * E(Y) + Cov(X, Y) \tag{4}$$

2.3 Monte carlo perft estimation

Monte carlo methods take random samples of a population to compute expected outcome of a phenomenon with a certain degree of certainity. Recently they have become quite popular in computer games such as Go where static evaluation of a position is very difficult. The Monte Carlo Tree Search (MCTS) method pioneered by Coulom [2006] combines monte-carlo simulations with a dynamically growing tree. It is the backbone of the strongest computer Go programs. This work discusses the use of similar methods for perft estimation and draws parallels between the two applications in the hope of gaining insight to further improvements. We will borrow terms from MCTS to differentiate selection policies in the tree part from those in the monte-carlo part. The former is termed as the *tree policy* and the later is the *default policy*.

2.3.1 Monte carlo for perft vs its other applications

Let us consider perft estimation using monte carlo simulations done from the root i.e. where the tree policy is absent. Random moves are made starting from the root to the depth perft is being computed. At the terminal position, some kind of *reward* is assigned to the simulation. It is insightful to look at the kind of rewards and back up operations done for other monte-carlo applications, so we will review them briefly.

1. In two player games such as Go, the reward is usually either a Win,Loss or Draw with numeric values of -1,0 and 1 respectively. The back up procedure follows Minmax operations therefore the root node receives exactly the same reward as the terminal position. For Negamax updates, the sign of the reward the root node receives depends what type of nodes (MAX or MIN) the root and terminal positions are. In any case the backup operation is still simple, but we will consider Minmax from here on for convenience.

2. With a depth limited monte-carlo search, the terminal position may not be a finished game position. In that case, a heuristic evaluation function can be used to compute infinitely many rewards in the range of [-1,1], or any other convenient range. Other than that, the backup procedure is the same as in the first case.

3. An interesting case is when the terminal position is assigned a random reward in the range of [0,1] and then combined with Minmax updates in the tree. It is known that a maximizing operation on random numbers gives a skewed distribution to the right, which is unlike averaging that is centered around the mean 0.5. Therefore a parent MAX node receives increasingly larger values depending on the number of leaf nodes it contains. Hence it can be considered as an indirect measure of mobility which is an important component of chess evaluation. This emergence of intelligences from what looked like utter randomness is informally known as the "Beal effect" after his inventor Don Beal.

4. Finally the problem we are investigating here, perft, gives rewards of 1. This indifference is rather strange compared to what happens in the above three cases. All of them give rewards based on characteristics of the terminal position but perft always gives a reward of 1 which is equal to perft(0). If we follow the same update procedure as above, i.e. Minmax, for this case as well, it implies the monte-carlo simulations become useless since all nodes will receive a score of 1 anyway.

The interesting aspect of perft estimation is that the reward is actually contained in the path traversed and not in the terminal position. Therefore reward received at the root node is path dependent. This makes perft estimation unique compared to the rest.

2.3.2 Proof of convergence and unbiasedness

Consider the set of leaf nodes S whose cardinality $|S|$ is the value of perft. Each element has a probability p_i of being selected that depends on the path that must be taken to reach it from the root. Assuming the probability of selection of internal nodes at different plies are independent of each other then p_i is the product of probabilities of selection of nodes encountered on the way to the leaf node p_{jk}.

$$p_i = \prod_{j=0}^{d} p_{jk} \tag{5}$$

The inverse of this probability $\frac{1}{p_i}$ is actually an estimate of perft, and it is the reward assigned to the simulation whenever the leaf node is selected. Conducting multiple simulations with this reward should therefore give an unbiased estimate of perft $|S|$ as shown below.

$$\begin{aligned} X_i &= \tfrac{1}{p_i} \\ \mu_X &= \sum p_i X_i \\ &= \sum p_i(\tfrac{1}{p_i}) \\ &= |S| \end{aligned} \tag{6}$$

This is a convenient way of proving that the simulations will lead to unbiased estimates of perft. The original proof by Osterlund P. uses product and sum operations at internal nodes, which we have masked by equation 5. It is to be noted the assumption of independence of selection probabilities at different plies is not necessary for the proof, and is done for convenience.

2.3.3 Random pruning method

The first to use the monte-carlo method in the perft betting competition was Muller H.G. The method he used was to accept moves with a certain probability p_i, and then multiply the total count by $\frac{1}{p_i}$. Therefore with this method two or more moves can be accepted at a ply. If no moves fail in the acceptance range, then the sub-tree will be pruned completely. Later on he modified his method such that acceptance is made on a per-move basis.

Then from from the number of moves accepted, $p_i = \frac{N_{acc}}{N}$ is calculated after which the same procedure is followed. Tapering the acceptance probability with depth, i.e one that starts with p_i=1 at ply =0 and converges to p_i=0 at the perft depth, can be used to improve the performance with out a lot of effort. It is most likely that Labelle [2007] used exactly this same method as Muller's because of ease of implementation. Later on Osterlund P. modified this method to accept exactly one move at each ply, whose details we have discussed in the previous sections. The result was a lower standard error per number of move generation calls, which at the time were found convenient for comparing performance. However considrering more moves in the upper parts of the monte carlo part could be beneficial, specially when the tree is non-uniform.

2.3.4 Monte-carlo perft by darts

Another monte-carlo method used by Blass Uri has some similarity with the classical monte-carlo method to determine π by throwing darts. It is less efficient than the other methods due to the potential of playing many illegal games, however it is an interesting approach to analyze. First upper bounds on the branching factors $U(i)$ are determined by some means. Then a random number with in the limit of the upper bound is generated to pick a move. If the move falls in the range of legal moves, it is played on the board and the game continues. Otherwise the sequence is aborted and the game is counted as illegal. Then the unbiased perft estimate can be found from the fraction of legal games played and the upper bound perft estimate obtained by multiplying $U(i)s$.

$$Y = p(legal) \prod_{i=0}^{d} U(i) \tag{7}$$

2.3.5 Sampling distribution

Let us first look at the underlying distribution when monte-carlo simulations are begun right from the root i.e. full width parameter fwd set to 0. In this case each perft estimate (sample) is a product of branching factors suggesting a log-normal sampling distribution. Indeed that turns out to be the case as shown in figure 3. If $fwp = 1$, then each sample will contain 20 samples of the previous type from the 20 moves at the root. Thus the sampling distribution will start to look like normal distribution as a result of the central limit theorem. The sampling distribution can be assumed to have reached full normality for $fwd \geq 2$.

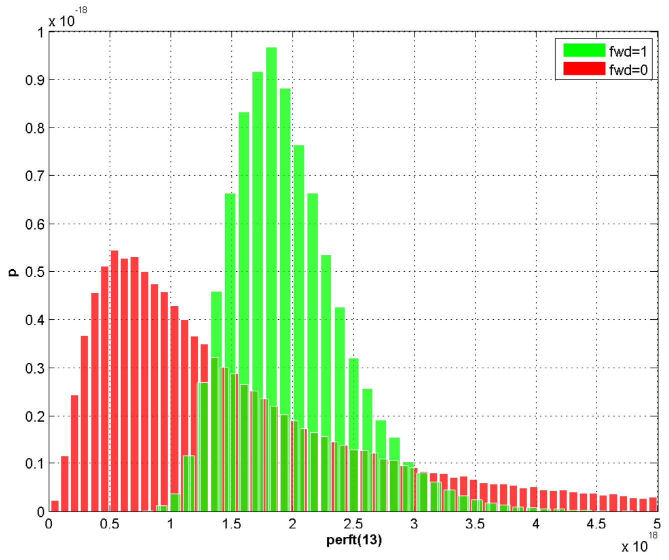

Figure 3: Sampling distributions

2.3.6 Quantifying simulation efficiency

One can not compare samples taken with different values of fwd directly. For instance, to measure the effect of fwd on variance reduction, the number of samples taken should be proportional to perft(fwd). A better approach to compare different variance reduction algorithms is suggested in Haugh [2004]. Simply put, W is more efficient than Y if and only if

$$Var(W)E_w < Var(Y)E_y \tag{8}$$

where E_w and E_y are work required to compute one sample of W and Y respectively. Michel Van den Bergh suggested that this work be proportional to the number of move generation calls. Generating moves and testing for legality is supposedly the most time consuming part of perft. Thus we will use what will henceforth be referred to as "cost" to compare algorithms

$$Cost = SE * \sqrt{N_{moves}} \tag{9}$$

Simulations must be carried out until this parameter stablilizes as shown in figure 4.

Table 1 shows that with equal number of "equivalent samples", the performance with different fwd are comparable, unlike the case if we used 1.6M samples for all. For $fwd \leq 2$, there is not a lot of improvement, but a clear

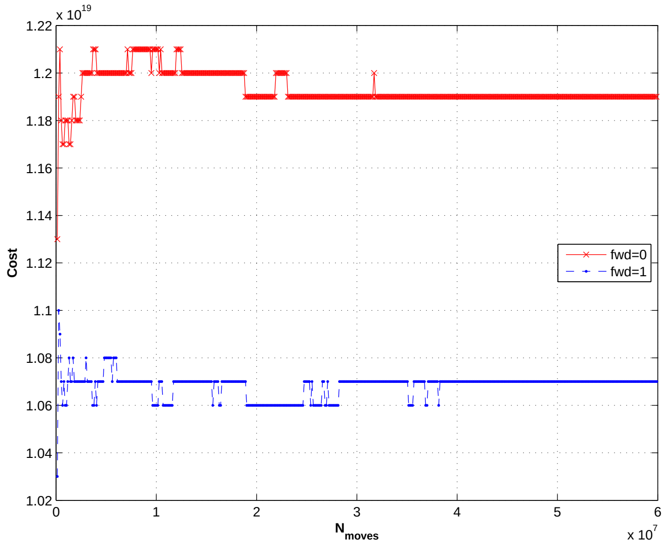

Figure 4: Cost graphs

jump in performance is observed at $fwd = 3$. One might wonder why there is even a performance increase at all going from $fwd = 0$ to $fwd = 1$. All the 20 root moves take equal number of simulations in both cases, so it is not obvious where the improvement is coming from. The tree expansion helps us do what is known in statistics as *stratified sampling*. Here the root moves are used to classify the simulations resulting in reduced overall variance. Stratified sampling achieves this in two ways

1. Strata help to reduce variance due to the restriction on where samples are taken. Free selection of samples and then summing will result in more variance than always taking one sample from each node followed by summation.

2. Non-uniform allocation of simulations proportionate to standard deviation gives minimum variance as will be discussed section 2.4.3.1.

We do not have the second advantage for $fwd \leq 2$, so the observed improvement must come from the first advantage. But starting from $fwd = 3$ moves such as *e2e4* start to take more simulations than others. Hence the large increase in performance can be attributed to non-uniform allocation of simulations. In addition, even if we made sure to take equal number of samples for all tests, the number of move generation calls are not equal. This resulted in lower cost going deeper. It is easy to see the underlying reason if

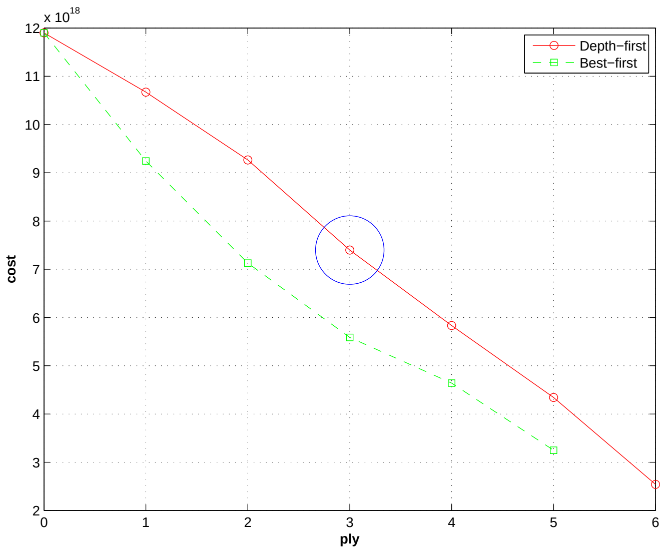

Figure 5: Decrease of cost with depth

we imagine the tree as a trapezium with top width of 20 moves and bottom width of perft(fwd) leaf nodes. If monte-carlo simulations carried out right from the root, the shape will be a rectangle. Therefore whatever doesn't fall in the trapezium can be considered as the additional cost. This difference is more pronounced when the tree is pre-generated and stored in case of a best-first perft estimator. Move generation in the tree is done only once in that case, thereby lowering the cost of perft estimation significantly especially at small depths. In conclusion cost is a better performance measure of simulation efficiency than number of simulations.

There is yet another significant drop in cost at $fwd = 6$ as shown in figure 5 but the number of samples taken were few to make any solid conclusions. This is due to use of unoptimized perft program which takes a long time to compute perft(6) which has about 119 million leaf nodes. The graph also shows results obtained by using a non-uniform selection policy using a best first perft estimator. A significant drop of cost for the best-first estimator compared to the depth-first estimator is observed up to perft(5). It was not possible to store all perft(6) positions in memory, so comparison is not done at that depth.

Table 1: Effect of full width parameter on perft(13) estimation

fwd	Samples	N_{moves}	SE	ΔSE	Cost	$\Delta Cost$
0	1600000	39990511	1.886e15		1.193e19	
1	80000	36870687	1.753e15	5.078e14	1.064e19	1.455e18
2	4000	33675040	1.616e15	5.120e14	9.380e18	1.410e18
3	180	30511165	1.289e15	1.053e15	7.119e18	2.282e18

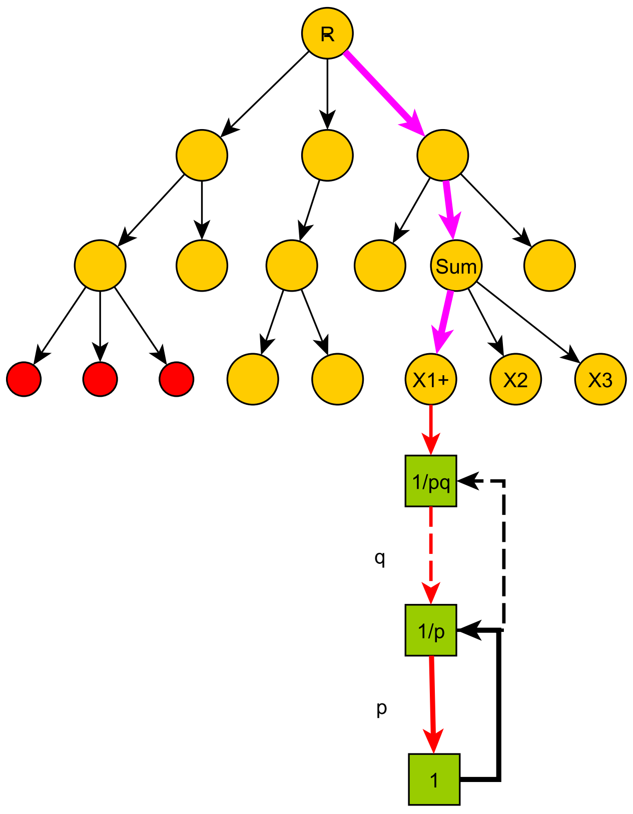

Figure 6: Monte-carlo perft tree showing selection,expansion,simulation and back-propagation steps

2.4 Best-first perft estimators

In the previous section we looked at depth-first perft estimators. Now we turn attention to best-first perft estimators which store part of the tree in memory. This version is what is commonly implemented in MCTS, even though depth-first variants are also used in some cases. It can be implemented either by the use of transposition tables or doubly-linked list data structure. Regarding perft, the best-first approach has the following advantages

1. The simulations could be allocated in an optimal way. In monte carlo Go programs, the upper confidence bound (UCB) formula is usually used to guide the search to the most promising lines. It balances search for new best move (exploration) with selection of the move that leads to best results so far (exploitation). It is not clear what the motivation for exploration is in case of perft. A possible motive is that some nodes whose estimates happen to be under-estimated during the initial iterations will improve their estimates if given the chance. However greedy selection of nodes that leads to largest reduction in variance seem to work well in practise.

2. The tree can be dynamically grown which could prove to be a significant advantage for selective trees. In MCTS expansion of trees is done towards lines that lead to best play. The same idea can be applied in perft, by searching deeper along bigger subtrees.

3. Because the tree is stored in memory, moves are generated only once at start up. This reduces the cost of perft estimation significantly and will give edge to the best-first estimator, even when the same selection policies are used.

The four major steps of MCTS are selection, play-out, expansion and back-propagation.

2.4.1 Back propagation

The result of monte-carlo simulations, which in the case of perft is contained in the path, is propagated backwards to the root. In the monte-carlo part, the back propagation strategy is to multiply the estimated perft of the child node with the probability of the path taken to select it as depicted in figure 6.

$$Y_i = X_i(\frac{1}{p_i}) \tag{10}$$

Inside the tree, the back propagation is done by adding the mean perft estimates of the child nodes. The associated uncertainty is also a sum of uncertainties in the children nodes. Both operations are done after every simulation.

$$\begin{aligned} \mu_Y &= \sum \mu_{X_i} \\ \sigma_Y &= \sum \frac{\sigma^2_{X_i}}{N_i} \end{aligned} \tag{11}$$

2.4.2 Default play-out policy

Consider the case where estimate of the true perft $\sum x_i = x_1 + x_2 + ... + x_m$ is made from simulation result X_i of exactly one move. The perft estimate is then calculated as $X_i(\frac{1}{p_i})$. For uniform selection policy, the multiplying factor $\frac{1}{p_i}$ is always the number of legal moves m. If the default selection policy is non-uniform, then moves selected with small probability will have larger multiplying factors greater than m and vice versa.

Proof. Default policy optimum weights are proportional to sub-tree sizes

$$\begin{aligned}
Y_i &= \tfrac{X_i}{p_i} \\
\\
\mu_Y &= \textstyle\sum p_i Y_i \\
&= \textstyle\sum p_i \tfrac{X_i}{p_i} \\
&= \textstyle\sum X_i \\
\\
\sigma_Y^2 &= \textstyle\sum p_i (Y_i - \mu_Y)^2 \\
&= \textstyle\sum p_i (\tfrac{X_i}{p_i} - \sum X_i)^2 \\
&= \textstyle\sum p_i (\tfrac{X_i^2}{p_i^2} - \tfrac{2X_i \sum X_i}{p_i} + (\sum X_i)^2) \\
&= \textstyle\sum [\tfrac{X_i^2}{p_i} - 2X_i \sum X_i + p_i (\sum X_i)^2] \\
\\
\text{Minimize:} &\quad f(p_1, p_2, ..., p_m) = \sigma_Y^2 \\
\text{Subject to:} &\quad g(p_1, p_2, ..., p_m) = \textstyle\sum p_i = 1 \\
\\
\Lambda(p_1, p_2, ..., p_m, \lambda) &= f(p_1, p_2, ..., p_m) + \lambda \cdot \Big(g(p_1, p_2, ..., p_m) - 1 \Big) \\
&= \sigma_Y^2 + \lambda \cdot \Big(\textstyle\sum p_i - 1 \Big) \\
&= \textstyle\sum [\tfrac{X_i^2}{p_i} - 2X_i \sum X_i + p_i (\sum X_i)^2] + \lambda \cdot \Big(\sum p_i - 1 \Big) \\
\\
\nabla \Lambda(p_1, p_2, ..., p_m, \lambda) &= (\tfrac{\partial f}{\partial p_1}, \tfrac{\partial f}{\partial p_2}, ..., \tfrac{\partial f}{\partial p_m}) \\
\\
\tfrac{\partial f}{\partial p_i} &= \tfrac{-2X_i^2}{p_i^2} + (\textstyle\sum X_i)^2 + \lambda \\
\\
p_i^2 &= \tfrac{2X_i^2}{(\sum X_i)^2 + \lambda} \\
p_i &\sim X_i \\
p_i &= \tfrac{X_i}{\sum X_i}
\end{aligned} \tag{12}$$

□

2.4.3 Tree selection policy

2.4.3.1 Optimum allocation of simulations Let us now consider the selection process in the tree where monte-carlo simulations are conducted for each of the moves and the results are summed to get perft estimate for the parent node. This is slightly different from what is done in the monte-carlo part because there the estimate is made from the result obtained from exactly one move. The question we want to answer here is given a total of N simulations to spend on all moves, what is the best way to divide the simulations among them for minimum cumulative variance? This is a similar optimization problem as before, except that number of simulations N_i for each move is required instead of p_i. Suppose the perft counts (sub-tree sizes) are μ_{X_i} and the population variances are $\sigma^2_{X_i}$. Perft is a sum operation thus the mean and variance of the perft estimate for the root node are sums of the mean and variances in each child nodes.

Proof. Tree policy optimum weights are proportional to standard deviation

$$\begin{array}{rcl}
\mu_Y & = & \sum \mu_{X_i} \\
\sigma_Y & = & \sum \frac{\sigma^2_{X_i}}{N_i} \\
\\
\text{Minimize:} & & f(N_1, N_2, ..., N_m) = \sum \frac{\sigma^2_{X_i}}{N_i} \\
\text{Subject to:} & & g(N_1, N_2, ..., N_m) = \sum N_i = N \\
\\
\Lambda(N_1, N_2, ..., N_m, \lambda) & = & f(N_1, N_2, ..., N_m) + \lambda \cdot \Big(g(N_1, N_2, ..., N_m) - N\Big) \\
& = & \sum \frac{\sigma^2_{X_i}}{N_i} + \lambda \cdot \Big(\sum N_i - N\Big) \\
\\
\nabla\Lambda(N_1, N_2, ..., N_m, \lambda) & = & (\frac{\partial f}{\partial N_1}, \frac{\partial f}{\partial N_2}, ..., \frac{\partial f}{\partial N_m}) \\
\\
\frac{\partial f}{\partial N_i} & = & \frac{-2\sigma_i^2}{N_i^2} + \lambda \\
N_i & \sim & \sigma_i \\
N_i & = & N(\frac{\sigma_i}{\sum \sigma_i}) \\
p_i & = & \frac{\sigma_i}{\sum \sigma_i}
\end{array} \tag{13}$$

□

It is to be noted that this method is known in statistics as *stratified sampling* and the optimum allocation strategy is known to be proportionate to standard deviation for each stratum. The proof given by Michel Van den Bergh during perft betting competition is given here for convenience.

2.4.3.2 Greedy selection algorithm Let us look at the previous problem from a different perspective. Given N_i simulations already invested on each of the moves with the same population mean and variance as before, which move should be picked for the next simulation to get maximum variance reduction? The variances for each moves at the beginning are $Var(X_i) = \frac{\sigma_i^2}{N_i}$ and after one more simulation will be $Var(X_i)^+ = \frac{\sigma_i^2}{N_i+1}$. The difference is the amount of variance reduction we can get by conducting one more simulation on move i. Naturally we pick the move which gives the largest reduction in variance. Therefore the best move to pick (move k) is obtained as follows

$$\begin{aligned} k &\leftarrow \arg\max_i \left(Var(X_i) - Var(X_i)^+\right) \\ &= \arg\max_i \left(\tfrac{\sigma_i^2}{N_i} - \tfrac{\sigma_i^2}{N_i+1}\right) \\ &= \arg\max_i \tfrac{\sigma_i^2}{N_i(N_i+1)} \\ &= \arg\max_i \tfrac{Var(X_i)}{N_i+1} \end{aligned} \tag{14}$$

2.4.3.3 Probability matching methods It is possible to convert the optimal proportioning formula in equation 13 to a selection method. We select the next move to be simulated using equation 15 so that the selected move has the largest difference from the optimal ratio at any time during the simulations. This kind of selection algorithms are known in literature as *probability matching methods*. Similarly other probability matching methods can be derived by substituting the ratio on the left $\frac{\sigma_i}{\sum \sigma_i}$ with another reasonable ratio. For example mean perft estimates are usually proportional to the standard deviation thus a selection algorithm based on mean can be made by replacing that ratio with $\frac{\mu_i}{\sum \mu_i}$ as shown in equation 15. Similarly a uniform selection among M moves follow equation 16.

$$k \leftarrow \arg\max_i \frac{\sigma_i}{\sum \sigma_i} - \frac{N_i}{\sum N_i} \tag{15}$$

$$k \leftarrow \arg\max_i \frac{1}{M} - \frac{N_i}{\sum N_i} \tag{16}$$

2.4.3.4 Proof of equivalence The greedy selection algorithm and the optimal allocation method are practically equivalent. One can show this equivalence by performing simulations of the selection process for given standard deviations. The resulting sequence from both methods are the same, except in a few cases where a +1 difference is observed in the N_i. Despite its first look, the greedy selection algorithm leads to the same allocation scheme as the optimal. The matlab code below demonstrates the simulation process.

```
sigma = [32 16 8 4 2 1];
sumsig = sum(sigma);
n1 = [1 1 1 1 1 1];
n2 = [1 1 1 1 1 1];
for i=1:10000
    maxr=0;
    maxj=1;
    for j=1:6
        t=(sigma(j)^2)/(n1(j)*(n1(j)+1));
        if(t>maxr)
            maxr=t;
            maxj=j;
        end
    end
    n1(maxj)=n1(maxj)+1;

    sumn2 = sum(n2);
    maxr=-1;
    maxj=1;
    for j=1:6
        t=(sigma(j) / sumsig) - (n2(j) / sumn2);
        if(t>maxr)
            maxr=t;
            maxj=j;
        end
    end
    n2(maxj)=n2(maxj)+1;
    disp(n1-n2);
end
```

2.4.3.5 The under-estimation problem A straight forward implementation of the optimal tree policy soon reveals a problem. The selection of nodes for simulation depends directly or indirectly on the sub-tree sizes. This is true for both the optimal proportioning scheme that uses standard deviations and the one that uses mean sub-tree size. In practice both give similar results and sometimes the mean sub-tree size proportioning scheme may be better at lower number of simulations. The reason for underestimation is as follows. Say a larger sub-tree size node is picked for simulation, and if after a couple of simulations the mean drops then another larger node is selected and so on. The effect is that the selection scheme tends to minimize the overall perft at any instant. It gives one the idea that the estimate may be biased, but in reality it is converging at a very slow rate. The behavior can be reproduced with a tree that sums product of random numbers at the leaf nodes, that shows that the problem is not specific to perft estimation. Hence our effort to reduce the variance is affecting the location of the mean. Solutions to this problem are as follows

1. One can use fixed allocation schemes that do not change selection behavior by looking at simulation data. Uniform and expanded sub-tree size allocation schemes are examples that fix this problem. The latter will make the best-first estimator to be exactly same as a depth first estimator except for the saving in move generation in the tree.

2. Using an expanded sub-tree size allocation scheme only in the frontier and pre-frontier nodes practically fixes the problem but the cost will increase due to use of sub-optimal selection policy there. In the previous sections we have seen that the sampling distribution is log-normal and it takes a ply or two to change to a normal distribution. Hence having a transition zone that does that reduces the problem significantly but it will not make it completely go away due to the use of non-uniform selection policy in the upper plies.

3. The preferred solution is to use a second sequence of perft estimation that is done much less frequently than the primary estimator. With sub-tree size proportioning scheme even a 1 in 100 call of the secondary estimator gives good results, so the cost of the secondary estimator can be kept very low. The primary estimator makes selection judgments based on the secondary estimators results and this completely avoid the problem of under-estimation while barely increasing the cost.

2.4.4 Expansion

When a leaf node is visited enough number of times, the tree is expanded by adding its child nodes. The strategy currently used is to discard the results of all the previous simulations at the parent node and carry out new ones starting from the child nodes. The number of simulations should be equal to or more than the previous simulations so that the expansion does not result in more uncertainty than it was before. Without this precaution, the tree will be expanded narrowly along one line if the selection strategy is to select nodes in proportion to their standard deviation.

The effect of expansion is that one can start simulations from the root and the tree grows dynamically along lines with bigger sub-tree sizes. For unbalanced or selective trees, this could prove to be very important. Previous tests for the best-first approach were all done with pre-generated trees at the start of the simulations. This is more efficient but only because the start position of chess is more or less balanced with the exception of few moves with larger sub-trees. With a dynamically grown tree, it is possible to get perft estimations at a slightly higher cost than pre-generated trees for the same depth. However dynamically grown trees are more robust, because the tree learns which way to grow itself.

2.5 Depth-first in hindsight

Consider a depth-first perft estimator that starts to do monte-carlo simulations at some shallow depth r different from 0. On each call of perft(d), the leaf nodes at ply=r are visited exactly once. We have shown in the previous sections that non-uniform selection is optimal, but at first glance the depth-first searcher seems to do a uniform selection. However during the perft competition this method proved hard to beat, even though it was eventually out performed. The reason for this was not obvious at first, but is easy to understand on post-mortem. The tree selection policy is actually non-uniform that depends on the expanded sub-tree sizes i.e. up to a depth of r. The visits at the internal nodes are inherently proportioned by an proportioned estimate of the real sub-tree sizes which we now know is a good way to reduce variance. The effect is not visible for depth of expansion less than 2 because each of the moves have 20 replies, thus making it a uniform selector. At $ply = 3$ there is a performance jump since moves that are known to lead to larger sub-tree sizes such as e2e4 start to have larger expanded sub-tree sizes.

However the method fails badly when the tree is selective such as when using reductions and extensions, or even on other start positions that naturally lead to unbalanced trees. In general when the expanded sub-tree sizes vary significantly from the actual sub-tree sizes up to the perft depth, the method performs poorly. The initial chess position has more or less a balanced tree but even then it was possible to show improvements from allocating simulations by measured sub-tree sizes. It is easy to prove that the method can be beaten. If one does an r ply full width followed by monte-carlo simulations, then one can also pre-generate a tree to a depth s and then apply $r - s$ full width followed by monte-carlo simulations. It is known that sub-tree size proportioning is not the best so we can use a better method for tree policy to beat it. Also best-first estimators incur less cost due to moves being generated once, thus this helps to widen the gap further.

2.6 Other enhancements

2.6.1 Default policy heuristics

A uniform selection policy is not optimal for least variance, hence it can be improved by using domain dependent knowledge similar to other MCTS applications. The best selection strategy is shown to be proportioning by estimated sub-tree sizes. One can use heuristics to assign probabilities for a move to be selected using a roulette wheel selection algorithm. For instance, it is known that captures of pieces lead to smaller sub-trees than other moves, therefore the probabilities are reduced accordingly. Moves of pieces that lead to better mobility (e.g. e2e4) can be assigned higher probabilities. Similarly other simple heuristics that do not hurt performance can be used to bias selection of moves towards larger sub-trees. We recall that the perft estimate will be of similar magnitude whether it came from a move with larger or small sub-tree size.

2.6.2 Antithetic variates

We can further improve the default selection policy by conducting two monte-carlo simulations instead of one. This is known as the method of antithetic variates for variance reduction. The basic idea is to pick the second sample in a pre-determined way so that it has a negative correlation with the first sample. Hence the variance of the perft estimate will be smaller than if the variables were independent or positively correlated.

$$\begin{aligned} \hat{\theta} &= \tfrac{X_1+X_2}{2} \\ Var(\hat{\theta}) &= \tfrac{Var(X_1)+Var(X_2)+Cov(X_1,X_2)}{4} \end{aligned} \tag{17}$$

To benefit from this method, the moves are first scored using heuristics such as captures, piece square tables and history heuristics. Then they are sorted to apply a roulette wheel selection scheme. If the first sample is selected with probability p_i, the second anti-correlated sample is selected with probability $1 - p_i$. This method helped to reduce cost of perft estimation when applied at the leaf nodes in one layer. Considering more moves at the same or different plies did not help to reduce variance.

2.6.3 Parallelization

Monte carlo methods are usually suitable for parallelization, and that is the case for depth first perft estimators as well. Those methods are embarrassingly parallel because samples are taken only at the root from which statistics are computed. Thus one can start as many instances of the estimator as required with different random number seeds. This simple *root parallelization scheme* is also effective for best-first estimators that generates the tree at start up and never expands afterwards. When a uniform or expanded subtree size selection policy is used, the behavior is exactly same as that of depth first estimators'. The results using the optimal allocation scheme of proportioning by standard deviation are also good in practice, even though the allocation scheme depends on the data collected. A possible explanation is that a close to optimal allocation of simulations among internal nodes will be established quickly. From then on it becomes a case of collecting more data to bring down variance. However when expansion of the tree is allowed, other parallelization schemes may perform better. On shared memory systems, parallelization can be done either at the leaves, root or anywhere in the tree. The leaf parallelization scheme can be implemented by starting two or more threads to conduct simulations simultaneously. Antithetic variates nicely fit to this parallelization scheme but performance may be affected since threads have to wait for each others completion of simulation. A method that does not incur this penalty uses parallelization in the internal nodes with local mutexes. In conclusion parallel perft estimation using complex algorithms may not be worth the effort but it can serve as a playground for shared memory or cluster parallelization of MCTS.

3 Acknowledgement

This work would not have been possible without the inputs of those who participated in the informal perft betting competition. Thanks goes to the following people whose contributions I have not discussed in detail: Muňoz J., Schüle S., Scharnagl R. and Marcel J.

References

E. Anthony. The inexhaustability of chess. *The chess player's chronicle*, 2, 1878.

R. Bean. Counting nodes in chess. *Institute for studies in theoretical physics and mathematics*, 71, 2003.

A. Bertilson. Distributed perft project, 2011. URL `http://www.albert.nu/programs/dperft/default.asp`.

R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. *5th International Conference on Computer and Games*, 2006.

R. Eckhardt. Stan ulam, john von neumann, and the monte-carlo method. *Los alamos science, special issue*, 15, 1987.

S. Edwards. Perft(13) betting tool, July 2011. URL `http://talkchess.com/forum/viewtopic.php?t=39678`.

S. Edwards. Number of possible chess games at the end of the n-th plie., April 2012. URL `http://oeis.org/A048987`.

H. Haugh. Variance reduction methods. *Monte carlo simulation*, 2004.

F. Labelle. Statistics on chess games, 2007. URL `http://wismuth.com/chess/statistics-games.html`.

P. Osterlund. Perft(14) estimates, April 2013. URL `http://talkchess.com/forum/viewtopic.php?topic_view=threads&p=513308&t=47335`.

R.C. Smith. Program written as chess buff's research aid. *Compute world magazine*, April 1978.