Win/Loss States: An efficient model of success rates for simulation-based functions

Jacques Basaldúa and J. Marcos Moreno Vega

Abstract— **Monte-Carlo Tree Search uses simulation to play out games up to a final state that can be evaluated. It is well known that including knowledge to improve the plausibility of the simulation improves the strength of the program. Learning that knowledge, at least partially, online is a promising research area. This usually implies storing success rates as a number of wins and visits for a huge number of local conditions, possibly millions. Besides storage requirements, comparing proportions of competing patterns can only be done using sound statistical methods, since the number of visits can be anything from zero to huge numbers. There is strong motivation to find a binary representation of a proportion signifying improvement in both storage and speed. Simple ideas have difficulties since the method has to work around some problems such as saturation. Win/Loss States (WLS) are an original, ready to use, open source solution, for representing proportions by an integer state that have already been successfully implemented in computer *go*.**

I. Introduction

The ideas described in this paper have been developed for a computer *go* [1] playing program, but are applicable in many fields and no knowledge about the game of *go* [2] is required to read it. Monte-Carlo Tree Search (MCTS) has produced an unprecedented performance increase in the recent years in computer *go*. Based initially on the UCT algorithm [3], many advances have converted MCTS in the most successful *go* playing algorithm. To name a few: RAVE [4], progressive bias [5] and progressive widening [6]. Also, the tree search may benefit with additional specific evaluation in the opening [7]. In parallel with advances in tree search, simulated games played until no more moves are available (including some stopping rules), known as playouts, have also been an active research field.

Some approaches describe biasing playouts to follow simple *go* playing rules [8] and [9]; others describe offline learning of patterns from collections of master games [6] and [10] possibly with additional expert evaluation.

MCTS is an online learning algorithm that computes success rates of complete game states and compares those using confidence intervals, or some other heuristic, targeting compromise between exploration and exploitation. Since the tree is always limited to a number of nodes that cannot possibly represent the complexity of the entire game tree, when playout policies that do not learn are applied, moves no longer benefit of position specific knowledge. Ideally, playouts should benefit from some local context associated with the outcome of the move in terms of winning or losing the game. When the number of moves multiplied times the number of local contexts is bounded and fits in memory, it behaves like a "loose tree" extending the benefits of knowledge-driven MCTS-like behavior to the playouts.

Many researchers have described implementations that use online learned knowledge in the playouts. [11] describes the benefits of re-using the last good reply and [12] describes even further improvement from erasing the (state, action) combinations previously classified as last-win that have been observed again and no longer win. In our terminology, we call these approaches binary in the sense that a (state, action) combination has only two states: successful/losing (three, including never seen). This is equivalent to a Win/Loss State (WLS) with end of scale e=1. Other authors have implemented the use of complete (wins, visits) counters in the playouts, extending RAVE to a pool of n moves called PoolRave [13]. We name these applications (wins, visits)-based.

It is not the aim of this paper to describe a specific implementation. Our group has already implemented WLS in computer *go* and obtained over 50 Elo points improvement in the first experiments [14] implementing WLS in the simulated games known as playouts. The improvement is obtained from comparing a learning policy vs. a reference non-learning policy. The learning policy uses WLS. We are currently working together with a different team on implementing WLS in "learning playouts" in two different programs. Such description requires the extension of a research paper by itself. The current paper is a fully self-contained and domain independent description of WLS. It is necessary for understanding the ideas behind it and how the possible problems have been worked out. The open source code for implementing WLS in C++, Java and Pascal can be found at [14].

II. Win/Loss states

A. Definitions

Formally, a WLS with end of scale e is the set S of all

J. B.. Author is from: Departamento de Estadística IO y Computación, Universidad de La Laguna, 38271 La Laguna, Santa Cruz de Tenerife, Spain, (e-mail: jacques@dybot.com)

J. M. M. Author is from: Escuela Técnica Superior de Ingeniería Informática Universidad de La Laguna, 38271 La Laguna, Santa Cruz de Tenerife, Spain, (e-mail jmmoreno@ull.es)

proportions $s = n/m$ where $n, m \in \mathbb{N}, 0 \leq n \leq m$ and $0 \leq m \leq e$ over which three functions are defined: $v(s) \to \mathbb{R}$, $W(s) \to S$ and $L(s) \to S$.

The function $v(s)$ is defined over all elements other than $0/0$ and is a real value measuring the evidence that the proportion is above or below a given threshold value. It defines a total order over the set. The functions $W(s)$ and $L(s)$ represent the next state after a win and a loss respectively.

Many choices for the functions $v(s)$, $W(s)$ and $L(s)$ are possible, but we will only consider those fitting the following conditions:

Condition 1: $v(s)$ represents the confidence that a proportion s is above or below some reference proportion s_0 (by default, $s_0 = 1/2$). For this, we use the confidence interval for a binomial proportion using the lower bound (LB) $LB = \hat{p} - CI$ when the $s \geq s_0$ and the upper bound (UB) $UB = \hat{p} + CI$ when $s < s_0$. UB values are shifted by a constant to avoid overlapping between the LB and UB values. $v(s)$ is used only for sorting the set S and therefore, any value avoiding overlapping can be used. We chose the Agresti-Coull interval [15] and [16] because its continuity correction provides robust behavior when m is small. Since evidence increases as the number of visits increases: $v(3/4) < v(6/8) < \cdots$ and $v(1/4) > v(2/8) > \cdots$

$$CI = z_{1-\alpha/2}\sqrt{\hat{p}\,(1-\hat{p})/\hat{m}}$$

$$\hat{p} = (n + 1/2 \cdot (z_{1-\alpha/2})^2)/\hat{m}$$

$$\hat{m} = m + (z_{1-\alpha/2})^2$$

$z_{1-\alpha/2}$ is the $1-\alpha/2$ percentile of the normal distribution for given significance level α.

Condition 2: Updating s after a win always results in an increase of s except when s is already the supremum of S and the opposite applies to a loss. Formally:

$W(s) \geq s \quad \forall s \in S$ and $W(s) = s \Leftrightarrow s = sup(S)$
$L(s) \leq s \quad \forall s \in S$ and $L(s) = s \Leftrightarrow s = inf(S)$
(not including $0/0$)

Condition 3: Let $u(s) = (n+1)/(m+1)$ and $d(s) = n/(m+1)$:

$u(s)\, \epsilon\, S \Longrightarrow u(s) = W(s)$ and $d(s)\, \epsilon\, S \Longrightarrow d(s) = L(s)$

In other words, if the proportion resulting of adding one win or one loss to s is in S, the functions $W(s)$ and $L(s)$ will return that proportion. In other words, only the proportions with $m = e$ require some special attention. These proportions, called saturated, are studied below.

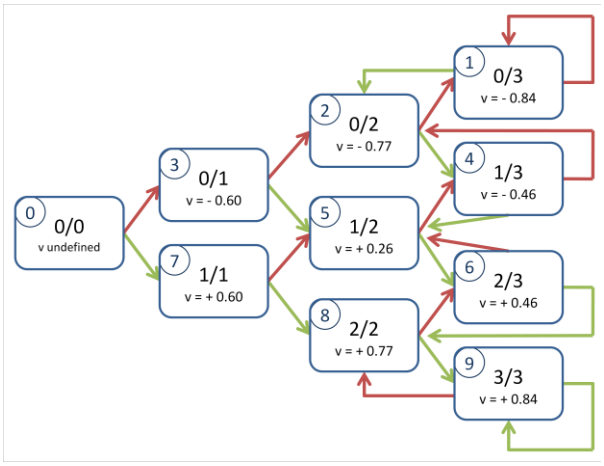

Fig. 1. A simple WLS with end of scale e = 3. Integers in the circles represent the binary value of each state following the order defined by v(s). Green arrows point to the next state after a win and red arrows point to the next state after a loss. Note that the proportions in the last column point backwards following the JPS heuristic.

B. *Implementation in a program*

A WLS is an integer that keeps track of the success rate of *(state, action)* pairs. The action is the move played and examples of items defining the state include: the color to move, the last move, the previous move, the number of liberties of adjacent groups, classified patterns formed by surrounding stones, etc. The possible combinations of input states can be very large. For example, on a 19x19 board, when the state includes 2 moves, the color to move and a local pattern identified in a collection of 100 classes, the resulting number of combinations exceeds 26 million. The amount of information updated is very large as well. If the implementation achieves 15K playouts/sec (a typical performance for an 8 core desktop), estimating 150 moves per playout on average and 30 minutes computing time for the whole game, 4 billion moves are generated during the whole game, each one updating a win or a loss to its combination of states. Therefore, compact representation and efficient integer operation are very important. Any WLS with end of scale $e \leq 21$ can be stored in 8 bits.

During the operation of the program, WLS management requires three basic operations: initializing, updating the state after a win or a loss and checking if the success rate is above some threshold. If it is, the corresponding move is played, possibly with a random choice when more than one candidates are found.

The WLS combination for all possible *(state, action)* pairs is stored as an array $s[\cdot]$ and i_{staction} is the index in the array identifying each pair.

Initializing is done by just clearing the array $s[\cdot]$. Zero represents the $0/0$ state, i.e., the *(state, action)* has not been observed previously.

Updating the WLS after a win or a loss is done via a Look-Up Table (LUT) update. Appropriate LUTs are created during the setup process and stored in some object wt. That class can be found in the source code [14]. After a win corresponding to the state i_{staction}, updating is just $s[i_{\text{staction}}] \leftarrow wt.WIN[s[i_{\text{staction}}]]$ and after a loss $s[i_{\text{staction}}] \leftarrow wt.LOSS[s[i_{\text{staction}}]]$.

Checking if a WLS is above some threshold is just comparing two integer values, since thresholds are converted to integer values in the setup procedure. A typical application will use the binary value corresponding to $1/2$ plus some configurable threshold. In the open source application the binary value corresponding to $1/2$ is stored in the field `wt.wls_binThresh_1div2`.

C. The setup procedure

The following pseudocode describes the setup procedure building the LUT tables as implemented in the open source code:

1. Create a table T with all proportions n/m where $0 \le n \le m \le e$. Each item t in T has four fields (n, m, v, i) only n and m are initialized in this step.

2. For each t in T evaluate $v \leftarrow v(n, m)$

3. Assign a value smaller than the smallest v to $v(0,0)$ and sort T in increasing v.

4. Assign an integer number i to each element in T starting with $i(0,0) \leftarrow 0$ and increasing by 1.

5. For each element $t(n, m, v, i)$

if $u(n+1, m+1, v_u, i_u) \in T$
then $WIN[i] \leftarrow i_u$
else $WIN[i] \leftarrow JPS(Won, n, m)$
if $d(n, m+1, v_d, i_d) \in$ T
then LOSS[i]$\leftarrow i_d$
else LOSS[i]$\leftarrow$JPS(Lost,n,m)

// Function JPS() is described below in IV.

6. Find the $t(n_t, m_t, v, i_t)$ corresponding to the threshold n_t/m_t and note its integer value i_t for its later use by the program.

III. THE SATURATION PROBLEM

A simple idea for updating a saturated proportion n/e could be:

$$WIN\ [n/e] = s(min(e, n+1)/e)$$
$$LOSS\ [n/e] = s(max(0, n-1)/e)$$

After the number of updated results is larger than e, the WLS behaves just like a counter increasing after a win and decreasing after a loss. This behavior is strongly biased towards 0 and 1. Bear in mind that any proportion above 1/2 produces more increase than decrease resulting in a state that will be near the top most of the time. And the same applies symmetrically.

To assess how well saturated WLS represent a proportion, we used the following setup: We took a set of 21 WLS measuring the output of 21 Random Number Generators (RNG) generating wins with a probability p_i of winning in equally spaced steps $p_i = \{0, 0.05, 0.1, \ldots 1\}$.

WLS are initialized at $s(0/0)$ and updated a number c of cycles with wins and losses depending on each RNG. The final state was converted back to a proportion $\hat{p}_i$ using the table T described in 2.C. $\hat{p}_i$ is an estimate of p_i.

We used two measures to assess the quality of $\hat{p}_i$:

$SD_r = \sqrt{1/(n-1) \cdot \Sigma_i (p_i - \hat{p}_i)^2}$ the standard deviation of the residuals and S_{rc} the Spearman rank correlation [17] between the set of p_i values and the set of $\hat{p}_i$ values, i.e., how well the order of the $\hat{p}_i$ values obtained experimentally represents the order of the original p_i values. We chose two measures, one focused in measuring absolute difference and the other focused in order, since in MCTS order is the most important. A better decision should ideally get higher evaluation than inferior decisions, which makes it explored before them; the value is not relevant.

Each complete experiment, consisting in updating the 21 WLS c times, was repeated 25000 times and the statistics were computed each time. We describe the distribution of SD_r and S_{rc} as $mean \pm SD$ (standard deviation).

TABLE I
SATURATION WITHOUT JPS HEURISTIC

Number of updates c	SD_r	S_{rc}
c=20 (not saturated)	*0.0899±0.0155*	*0.9624±0.0157*
c=200 (saturated)	*0.2316±0.0175*	*0.9170±0.0272*

Values of SD_r and S_{rc} obtained for $N = 25000$ experiments shown as $mean \pm SD$.

A nonsaturated WLS of $e = 21$ with $c = 20$ random updates is used as a control reference. Since $c < e$, it does

not produce any saturation. Residuals are just those expected by the randomness when $\hat{p}_i$ represents the count of observed wins and losses.

The increased error and the reduction in the Spearman rank correlation shown in table I both reveal that this idea has a problem with saturation.

IV. Jump-to Past State (JPS) heuristic. A solution to the saturation problem.

The saturation problem is produced because the policy in III makes extreme states $21/21$ or $0/21$ easy to reach by probabilities that are just above or below $1/2$, just because a counter that increases with a win and decreases with a loss will reach any arbitrary number when $p > 1/2$ given enough time. A solution to this is moving the next state to the past to make confirmation necessary before reaching the extreme state again. E.g., if a loss at the state $21/21$ goes to the state, $10/10$ (note that $v(10/10) < v(21/21)$) the WLS needs another 11 consecutive win updates to reach the state $21/21$ again. Also, a win at state $20/21$ should go back to force confirmation with a sequence of consecutive wins before state $21/21$ is reached.

This jump to the past must be longer for the extreme values than for values around 1/2. For the sake of simplicity, we tried a linear model that jumps more as the absolute difference to 1/2 increases. The maximum jump is defined by an empirically tuned constant K.

In pseudocode, the function JPS() used in 2.C is:

Function JPS(won : boolean; n, e : integer)

$$j = e - round(K \cdot e \cdot abs(n/e - 1/2))$$

```
if (won)
{
  if (n = e)
    then return the index of element e/e
    else return the index of the smallest n_j/j
         satisfying v(n_j/j) > v(n/e)
}
else
{
  if (n = 0)
    then return the index of element 0/e
    else return the index of the biggest n_j/j
         satisfying v(n_j/j) < v(n/e)
}
```

We tuned K with the same experiment described in III, but with the length of the saturated run randomized to avoid possible fitting to a specific number of times the end of scale is reached. Instead of using a constant $c = 200$, we used a uniform random $c \rightsquigarrow U(150,250)$. The problem starts to disappear as K approaches 1, i.e., the extreme states jump back to $e/2$. The best value found is $K = 1.3$.

TABLE II
Quality measures of saturated WLS with JPS heuristic

K	SD_r	S_{rc}
0.9	*0.1084±0.0147*	***0.9742±0.0105***
1.0	*0.0964±0.0141*	***0.9761±0.0098***
1.1	*0.0970±0.0140*	***0.9753±0.0095***
1.2	***0.0881±0.0132***	***0.9759±0.0095***
1.3	***0.0817±0.0125***	***0.9759±0.0097***
1.4	***0.0838±0.0129***	***0.9746±0.0102***

Values of SD_r and S_{rc} obtained for $N = 25000$ saturated experiments with a number of updates $c \rightsquigarrow U(150,250)$ and different values of K shown as $mean \pm SD$. Highlighted values show elements behaving better than a non-saturated WLS with c=20.

Note that all results shown in table II and highlighted are better than the results obtained by the reference WLS with $c = 20$ which is free of the saturation problem. This is consistent with the fact that $\hat{p}_i$ values estimated from approximately 200 RNG draws have smaller variance than those estimated from 20 RNG draws. For validation, identical results within 3 decimal digits were obtained by changing RNG seeds. The change from $c \rightsquigarrow U(150,250)$ to $c \rightsquigarrow U(1500,2500)$ also returned $K = 1.3$ as the optimal value with slightly smaller values of SD_r.

These results reveal that the JPS heuristic works out the saturation problem.

V. Step response

It is important to note that WLS have the feature of forgetting the past history of the success rates they measure when the probability of a win changes over time. For instance, when we are using results of past moves in a game, before we make a move, the playout policy learns a set of states. Then, a move is made and answered by the opponent. After that, the success rates of all other moves change, but most of the bad moves are still bad moves and most of the good moves are still good moves. Having approximate information may be better than having none. It is a decision of the program author whether to clear past information and start acquiring new more precise but less abundant information, or to keep the old information having more but less accurate knowledge. Using WLS we have a third possibility: merging past information with new information weighting the most recent events in a controllable way.

From a signal-processing point of view we can consider a WLS as a low pass filter and analyze its step response. Our study is just exploratory and a general procedure for computing the expected bias in $\hat{p}_i$ resulting from temporal changes in p_i is beyond the scope of this paper. But, we did empirically establish an interesting result: *The settling time is a linear function of the end of scale e*. This result reveals e as a tunable parameter that may have interesting applications, i.e., instead of always creating WLS with the maximum number of states for a given number of bits of storage, WLS with smaller e may benefit from smaller settling times in applications where that may be an advantage.

In our exploratory study, we only measured the settling time required by a WLS that has been measuring a probability $p = 1/4$ for a long time to reach $\hat{p} = 3/4$ within $\pm 2.5\%$ accuracy when p suddenly changes to $p = 3/4$.

We initialized the WLS at the state $1/4$ and generated a number of RNG draws with $p = 1/4$ to randomize the starting point. Then, we counted the number of updates required to measure $\hat{p} = 3/4$ within $\pm 2.5\%$ accuracy when generating updates with $p = 3/4$. The experiment was created for a series of WLS of different end of scale values e between 25 and 1000 in steps of 25. Each measure was obtained 25 times. A linear regression analysis between the average of the 25 measures and the e values shows a near perfect fit, with adjusted $R^2 = 0.99968$ and p-value $p < 10^{-5}$. The scatterplot is shown in figure 2.

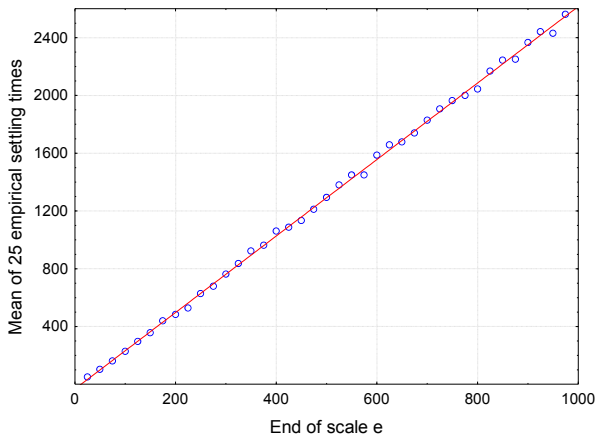

Fig. 2. A The scatterplot between the end of scale values and the mean of the empirical settling times measured from 25 repetitions

VI. Conclusions

Efficiency is the key for producing competitive game playing programs. Representing success rates by 8 bit integers instead of complete *(win, visits)* records which result in at least 8 times more space is an improvement when applications may need to store success rates for many millions of possible *(state, action)* pairs. Besides storage, at CPU speed WLS-based implementations outperform *(wins, visits)*-based ones. Updating a WLS is almost as fast as incrementing an integer variable; performing update via a LUT (which can be coded in a single XLAT CPU instruction) is one of the fastest possible operations. If the LUT is in CPU cache memory, which will happen when it is frequently used, the time is similar to updating one or two *(wins, visits)* counters. Furthermore, updating will not happen as frequently as finding new candidate moves. (Even in the minimal case in which only eight neighbors to the last move were considered as candidates, the number of comparisons vs. updates would be 8 to 1.) The latter requires checking if a proportion is above some threshold. Checking that a WLS is above some threshold is just an integer arithmetic instruction, while comparing *(wins, visits)*-based proportions requires computing a floating point confidence interval at the cost of many instructions. If that is not done for the sake of speed, the program will contain inherent flaws derived from unsound direct comparison between proportions based on much evidence and proportions based on just a few observations.

Since WLS is just a brick for building online learning simulation-based applications, prejudice about the lack of importance of doing things "just more efficiently" should be avoided. After all, MCTS is itself a success story about how hardware improvement enabled the possibility to explore ideas that would have been unfeasible one decade before they were implemented. Improvement in both storage and speed pushes the horizon of the "unfeasible" a little further away for new ideas to come. Also, WLS is not *go* specific and can be used in many other machine learning fields.

Acknowledgments

J. Marcos Moreno-Vega acknowledges the support from projects TIN2009-13363 (Spanish Ministry of Science and Innovation) and PI2007/019 (Canary Islands Government).

We also acknowledge Professor Peter Drake and his students from Lewis & Clarke College in Oregon for their valuable proofreading.

References

[1] M. Müller. "Computer Go", Artificial Intelligence, 134(1-2):145-179, 2002.
[2] "American Go Association: Go. Introduction to the game", http://www.usgo.org/what-go.
[3] L. Kocsis, C. Szepesvari, "Bandit based Monte-Carlo planning", in: 15th European Conference on Machine Learning, pp. 282-293, 2006.

[4] S. Gelly and D. Silver, "Monte-Carlo tree search and rapid action value estimation in computer Go", Journal Artificial Intelligence archive Volume 175 Issue 11, July, 2011.
[5] G. Chaslot, M. Winands, J. Uiterwijk, H. van den Herik, B. Bouzy, "Progressive strategies for Monte-Carlo tree search", New Mathematics and Natural Computation pages 343-357, 2008.
[6] R. Coulom, "Computing Elo Ratings of Move Patterns in the Game of Go", in H. J. van den Herik, M. Winands, J. Uiterwijk, and M. Schadd, ed., Proc. Computer Games Workshop 2007 (CGW 2007), pp. 113-124, 2007.
[7] J. Basaldúa, T.N. Yang and J.M. Moreno-Vega, "M-eval: A multivariate evaluation function for opening positions in computer go", Workshop on Computer Games (IWCG 2010), 2010.
[8] S. Gelly, Y. Wang, R. Munos, and O. Teytaud, 2006, "Modification of UCT with Patterns in Monte-Carlo Go", Tech. Rep. 6062, INRIA, France, 2006.
[9] Enzenberger, M., Müller, M. "Fuego an open-source framework for board games and Go engine based on Monte-Carlo tree search", Technical Report TR 09-08, University of Alberta, Edmonton, Alberta, Canada, 2009.
[10] Huang, S-C., Coulom, R., and Lin, S-S. "Monte-Carlo Simulation Balancing in Practice.", Computers and Games (CG 2010), 2010.
[11] P. Drake, "The Last-Good-Reply Policy for Monte-Carlo Go", ICGA Journal, vol. 32, no. 4, pp. 221-227, 2009.
[12] Baier, H. and Drake, P.D. "The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go", Computational Intelligence and AI in Games, IEEE Transactions on Issue, Volume 2, Issue 4 On page(s): 303 - 309, 2010.
[13] A. Rimmel, F. Teytaud, and T. Olivier, "Biasing Monte-Carlo Simulations through RAVE Values" in The International Conference on Computers and Games, 2010.
[14] J. Basaldúa, "WLS: open source implementation and complementary materials" http://www.dybot.com/WLS
[15] Agresti, A. "An Introduction to Categorical Data Analysis", Wiley, 1996.
[16] Brown LD, Cai TT, DasGupta A. "Interval estimation for a binomial proportion", Stat Sci (2001); 16:101-133, 2001.
[17] C. Spearman, "The proof and measurement of association between two things", Amer. J. Psychol., 15 (1904) pp. 72-101, 1904.