Evaluating the Complexity of Players' Strategies using MCTS Iterations

Pier Luca Lanzi
Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
pierluca.lanzi@polimi.it

***Abstract*—Monte Carlo Tree Search (MCTS) does not require any prior knowledge about a game to play, except for its legal moves and end conditions. Thus, the same MCTS player can be applied (almost) as it is to a wide variety of games. Accordingly, MCTS may be used as a touchstone to evaluate artificial players on different games. In this paper, we propose to use MCTS to qualitatively evaluate the strength of artificial players as the minimum number of iterations that MCTS needs to perform equivalently to the target player. We define this value as the "MCTS complexity" of the target player. We introduce a bisection procedure to compute the MCTS complexity of a player and present experiments to evaluate the proposed approach on three games: Connect4, Awari, and Othello. Initially, we apply our approach to compute the MCTS complexity of players implemented using MCTS with a known number of iterations, next to players using different strategies. Our preliminary results show that our approach can identify the number of iterations used by MCTS target players. When applied to players implementing unknown strategies, it produces results that are coherent with the underlying players' strength, assigning higher values of MCTS complexity to stronger players. Our results also suggest that, by using iterations to evaluate the strength of players, we may be able to compare the strength of algorithms that would be incomparable in practice (e.g. a greedy strategy for Connect4 and alpha-beta pruning for Awari).**

I. Introduction

Artificial players for board games come in a wide variety of flavors. Some implement basic greedy heuristics like for example "always do the move that captures the most pieces" in Mancala games [10]. Other ones apply traditional strategies taken from the game literature [10], [21], well-known search algorithms (e.g., A-Star, Iterative Deepening [15], Monte Carlo Tree Search [2], [6]), or more advanced approaches (e.g., a combination of Monte Carlo Tree Search and Deep Neural Networks [24]). These players are usually evaluated by comparing them against existing strategies in terms of performance. Published results usually demonstrate the superiority of one approach over another. The complexity of the players is typically discussed in terms of the time needed to perform one move or using a parameter connected to the algorithmic complexity of the player (e.g., the depth of the search for A-Star or the number of iterations for Monte Carlo Tree Search).

Monte Carlo Tree Search (MCTS) [7] is a decision making algorithm that has been successfully applied to a wide variety of games [4]. MCTS has several advantages over other well-known tree search methods. It performs an asymmetric tree growth that focuses on promising areas of the search space by visiting interesting nodes more often. MCTS is an anytime algorithm that can be halted at any time to return the current best estimate. Most importantly, for the goal of this paper, MCTS is aheuristic and does not require any prior knowledge about the domain except for its legal moves and end conditions. Thus, in principle, the same MCTS player can be applied, (almost) as it is, to a wide variety of games and may be used as a touchstone to evaluate artificial players on different games.

In this paper, we propose to use Monte Carlo Tree Search to provide a qualitative evaluation of the strength of an artificial player (implementing an unknown strategy) as the minimum number of iterations that MCTS needs to perform equivalently to such player. Figure 1 delineates a simplified version of the proposed scenario comprising an MCTS player using m iterations, a game with perfect information, a target player we want to evaluate, and a search algorithm. The goal is to compute the minimum number of iterations m^* that MCTS needs to match the performance of the target player. For this purpose, we first run n matches with the two players and collect the overall statistics such as the number of wins for each player, the number of ties, etc. If the statistics show that MCTS performs better than the target player, it means that the current number of iterations m is too high and should be decreased; conversely, if the statistics show that MCTS performs worse than the target player, the number of iterations m is too low and should be increased; when the two players perform equivalently (e.g., they both have the same win ratio or they tie most of the matches), the search stops and the current number of iterations is returned. Note that, since MCTS is stochastic (and the target player could be stochastic too), the procedure must be repeated several times to obtain a reliable estimate of m^* as, for example, the average over all the runs. The output of the procedure basically provides a qualitative evaluation of the strength of the target algorithm in terms of MCTS iterations since, *on average*, the target player performs *comparably* to an MCTS using m^* iterations. We name this value, the *MCTS complexity* of the target algorithm.

978-1-7281-1884-0/19/$31.00

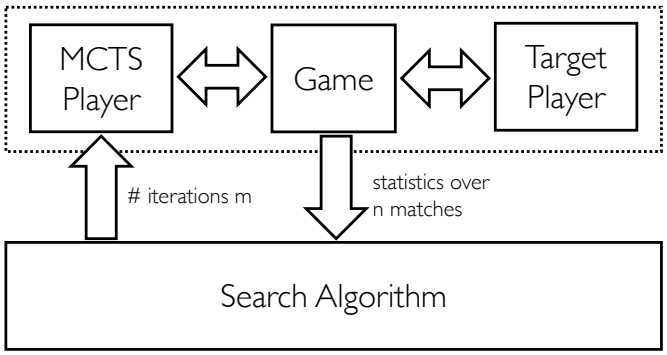


Figure 1: The proposed scenario.

The outcome of the above process is influenced by (i) the number of matches *n* that are run to compare the two players, the higher it is the more reliable the comparison is (as well as the computation time); (ii) the criterion applied to decide when two players perform comparably; and (iii) the algorithm used to update the current number of MCTS iterations based on the collected statistics.

In this paper, we present the results of a set of experiments we performed to provide a preliminary evaluation of the proposed approach. In particular, we considered three games, Connect4, Awari, and Othello. We applied the bisection method to search for the minimum number of MCTS iterations required to match the strength of the target player and we used two criteria to determine whether MCTS and the target player perform equivalently. The first criterion is based on the raw difference between the number of wins and thus defines players equivalence using a hard threshold. The second one applies True Skill [1] that characterizes players' skill as a normal distribution, and computes the quality of the match as an estimate of its draw probability. Initially, we apply bisection search to compute the MCTS complexity of players implemented using MCTS and a known number of iterations starting the search from given lower bound and upper bound. Next, we introduce a procedure to compute the initial bounds and finally apply bisection search to compute the MCTS complexity of players, available online, using different strategies. Our results show that our approach can identify the number of iterations used by MCTS target players and thus it can accurately evaluate MCTS complexity. When applied to players implementing unknown strategies, bisection results are coherent with the underlying complexity of the strategies, assigning higher values of MCTS complexity to stronger players. At the same time, by using iterations to evaluate the strength of players, we suggest we can compare the complexity of algorithms that would be incomparable in practice (e.g. an alpha-beta pruning for Connect4 against a greedy player for Othello).

II. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) identifies a family of decision tree search algorithms that has been successfully applied to a wide variety of board games [2], [6], [20], [25], card games [8], [9], [22], [26], video games [5], [18], and General Video Game Playing [11], [13], [19]. MCTS iteratively builds a partial and asymmetric decision tree by performing several random simulations. The algorithm comprises four steps: (i) selection, (ii) expansion, (iii) simulation, and (iv) backpropagation.

Selection: a tree policy is used to descend the current search tree by selecting the most urgent node to explore until either a leaf (representing a terminal state) or a not fully expanded node is reached; the tree policy typically uses the Upper Confidence Bound for Trees (UCB1 for Trees or UCT) [3], [14] to descend the child node that maximizes the UCT value computed as,

$$UCT(v') = \frac{Q(v')}{N(v')} + c\sqrt{2\frac{\ln N(v)}{N(v')}} \quad (1)$$

where v is the parent node, v' is a child node, $N(v)$ is the number of times v (the parent of v') has been visited, $N(v')$ is the number of times the child node v' is visited, $Q(v')$ is the total reward of all playouts that passed through v', and $c > 0$ is a constant that controls the amount of exploration.

Expansion: if the node does not represent a terminal state, then one or more child nodes are added to expand the tree. A child node represents a state reached by applying an available action to the current state.

Simulation: a simulation is run from the current position in the tree to evaluate the game outcome by following a default policy (typically random); this results in a reward computed according to a *reward function*.

Backpropagation: the reward received is finally backpropagated to the nodes visited during selection and expansion to update their statistics.

III. Evaluating Players' Strength with MCTS

MCTS does not require any strategic or tactical knowledge about the domain to make reasonable decisions. In principle, the same MCTS might be applied to a wide variety of games with perfect information almost as it is—for example, the exploration constant might require some tuning for the specific reward function and domain.

We propose to use MCTS as a touchstone to evaluate the strength of artificial players, implementing unknown strategies, in terms of the minimum number of iterations MCTS needs to perform comparably to such players. Figure 1 depicts the scenario we propose. To evaluate the target player, we run *n* matches between MCTS using *m* iterations and the target player. Based on the matches' statistics, the search algorithm adapts the number of iterations to find the value of *m* for which MCTS and the target player perform equivalently. In this

paper, we implemented the search algorithm using a bisection procedure that starts with a minimum and a maximum number of MCTS iterations (*min* and *max*). The initial number of iterations *m* is computed as the average of *min* and *max*. If the statistics show that the two players performed similarly, the process stops and the current number of iterations is returned; if the statistics show that MCTS outperformed the target player, the search moves to the lower half of the interval (identified by *min* and *m*); conversely, if the statistics show that the target player outperformed MCTS, the search moves to the upper half of the interval (identified by *m* and *max*). MCTS is a stochastic algorithm and the target player might be stochastic too. Accordingly, to obtain a reliable evaluation, we repeat the bisection process t times and output the value m^* computed as the average of $m_1 \dots m_t$. We name this value, the *MCTS complexity* of the target algorithm. The procedure assumes that the target player is weaker than MCTS using *max* iteration, if this is not the case, the procedure will return values near to the upper bound *max*.

A. The Search Algorithm

Algorithm 1 shows the bisection procedure in details. It takes as input the two players (*mcts* and *target*), a minimum and maximum number of iterations (*min* and *max*), and the number *n* of matches to perform to compare the players (line 1). At first, the upper bound and the lower bound for bisection search are initialized (lines 2-3), the current number of iterations is set to the average of the two (line 4), and the end condition is set to false (line 5). The following loop is repeated until the end condition is met (lines 6-26). The number of MCTS iterations is assigned (line 7) and *n* matches are played between the MCTS and the target player (line 8). If the match quality has been reached (line 9), the end condition is set to true otherwise the search boundaries are updated. If the MCTS player has won more matches (line 13), it means that the number of iterations used is too high and therefore the search should continue to the lower half of the search interval (lines 17-18); if the target player has won more matches, the number of iterations is too low and the search should continue to the upper half of the search interval (lines 17-18). If the two players won the same number of matches, the search should end.[1] Finally, the algorithm checks the updated search boundaries and if these are are too narrow the search stops (line 23).

B. Evaluating Match Quality

Bisection stops when either the bounds have become too narrow (Algorithm 1, line 23) or the two players have reached a comparable performance (Algorithm 1, line 9). We considered two measure of match quality (see Algorithm 2) that we applied to determine whether the players performed similarly, namely *Score Difference* and *True Skill*. The former computes the absolute difference *d* between number of wins of the players and compares it with a percentage θ_{sd} of the initial range (*max-min*): if *d* is smaller than θ_{sd}(*max-min*), the bisection stops. Score Difference provides a hard threshold to stop the bisection process; accordingly we considered a second criterion based on a probabilistic estimation of the players' skill. *True Skill* [1] is a skill-based ranking system developed by Microsoft Research. It is a generalization of the ELO rating system[2] used in Chess that can track the uncertainty about players' skills. It characterizes a player's skill as a normal distribution with mean μ, representing the player's perceived skill, and standard deviation σ, representing how uncertain the system is about the player's skill μ. True Skill maintains a belief in every player's skill using μ and σ: if the uncertainty is still high, the ranking system does not yet know exactly the skill of the player; if the uncertainty is small, the ranking system has a strong belief that the skill of the player is close to the average skill. By maintaining an uncertainty, True Skill makes big changes to the skill estimates early on but small ones after a series of consistent matches has been played. Given a belief of the players' strength, True Skill can compute the match quality as an estimate of the draw probability of the match, thus the higher it is the more similar the players' skill levels.

Algorithm 1 Bisection Search

```
1:  function BISECTIONSEARCH(mcts, target, min, max, n)
2:      lower ← min
3:      upper ← max
4:      iterations ← (lower+upper)/2
5:      end ← false
6:      repeat
7:          mcts.iterations ← iterations
8:          stats ← PlayGame(mcts, target, n)
9:          if MatchQualityReached(stats, min, max) then
10:             end ← true
11:         else
12:             diff ← stats.wins(mcts)-stats.wins(target)
13:             if diff>0 then
14:                 upper ← iterations
15:                 iterations ← (lower + upper)/2
16:             else if diff<0 then
17:                 lower ← iterations
18:                 iterations ← (lower + upper)/2
19:             else
20:                 end ← true
21:             end if
22:         end if
23:         if upper-lower≤threshold then
24:             end ← true
25:         end if
26:     until end
27:     return lower, upper, iterations, stats
28: end function
```

[1]Note that this condition would be captured by the MatchQualityReached function but we included this extra statement for completeness in the code.

[2]https://en.wikipedia.org/wiki/Elo_rating_system

We performed a set of simulations to compare the two criteria we adopted to evaluate the players' match results. For this purpose, we considered different probability values for winning and tying a game, simulated 1000 matches, and evaluated the results using Score Difference and True Skill. In our experiments, we used the implementation of True Skill taken from [17]. Figure 2, for each pair of tie and win probability values, reports (a) the score difference as a percentage of the number of matches played; (b) the match quality computed using True Skill. As can be noted, True Skill maintains a certain degree of uncertainty about its belief in players' skills even when players tie almost all the matches. In fact, True Skill values remains around 0.98 both when the tie probability is 1.0 (when we are certain that the players perform equivalently) and when the tie probability is just 0.7 and one player is winning 60% of the matches (when we might be not so certain that the players perform similarly). At the same time, when ties are less probable, True Skill appears to be quite effective at measuring the quality of matches. In contrast, Score Difference values (Figure 2a) seems to provide more information in borderline situations even when the tie probability is rather high. For example, with a tie probability of 0.8 or 0.9, the Score Difference values of similar winning rates (e.g., 0.5 and 0.4) are quite different (0.01 and 0.04 respectively). In the same situation, True Skill values are basically identical. These results suggest that Score Difference may provide a better measure to compare players' strength in games in which tying easily becomes highly probably (e.g., Tic Tac Toe)

Algorithm 2 Match Quality Evaluation

```
1: function MatchQualityReached(stats, min, max)
2:     if use_score_difference then
3:         d ← abs(stats.wins(mcts)-stats.wins(target))
4:         threshold ← θ_sd abs(max-min)
5:         return d≤threshold
6:     else
7:         return stats.tsquality≥ θ_ts
8:     end if
9: end function
```

IV. Validation with MCTS Players

We performed an initial set of experiments to validate our approach using players of known MCTS complexity on three games, Connect4, Awari, and Othello. Awari [16], [23] (also known as Wari, Owari, Awale, Awele, and Ayo) was introduced in 1991 by the computer scientists Victor Allis, Maarten van der Meulen, and H. Jaap van den Herik. It is a variation of the two-player abstract strategy Mancala board game Oware or Awele and it is the most widespread Mancala variant. All the games implemented the same reward function for MCTS that returns 1 for winning, 0 for losing, and 0.5 for tying; the UCT constant was set to 0.7, and applied the standard random simulation strategy.

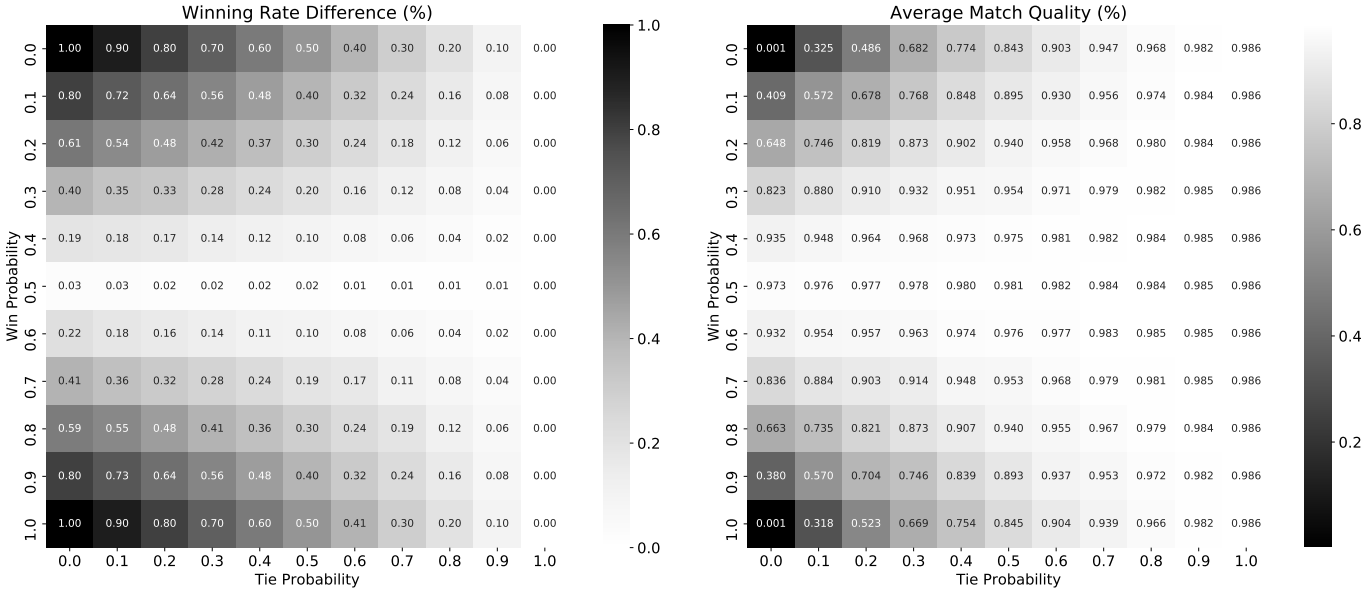
At first, we used bisection search (Algorithm 1) to evaluate Connect4 players implemented using MCTS with different number of iterations. The number of matches *n* was set to 1000; the initial bounds (*min* and *max*) were set to 1 and 2500 respectively; the Score Difference threshold was set to 1% of the initial range (θ_{sd} is 0.01) while the True Skill threshold θ_{ts} was set to 0.975. For each target player, we run ten trials and computed the average lower bound, upper bound, and estimated complexity. Then, we computed the ratio between these average values and the number of backups used by the target player. Figure 3 shows the results when using (a) Score Difference and (b) True Skill; values are averages over ten runs. The plots report the ratio of (i) the average upper bound ratio (solid circle markers), (ii) the average lower bound ratio (white circle markers), and (iii) the final number of backups (solid lines) with bars showing the standard error. As can be noted, bisection search using Score Difference (Figure 3a) accurately evaluates the target players. The ratio between the average number of backups returned by the search and the number of backups used by the target player is very close to one, apart from few exceptions (e.g., 200, 500, and 1400 iterations). The evaluation performed using True Skill is less accurate (several times the ratio is slightly farther from one but always below 1.1, that is below 10% of the target number of iterations) and more noisy (the standard error bars are larger).

We repeated the same experiment using Awari with the same settings. Figure 4 shows the results when the evaluation was performed using Score Difference (Figure 4a) and True Skill (Figure 4b). The plots confirm the results for Connect4. Quality evaluation based on Score Difference accurately evaluates the target players with ratios always close to one and small standard error values. True Skill results in less accurate evaluations, the ratio is often slightly larger than one and more noisy as the larger error bars show.

Finally, we applied bisection search to evaluate target players on Othello using a 6×6 board. Table I reports the target number of backups, the initial search interval, and the results produced by Score Difference and True Skill; values are averages over ten runs; mean and standard error are reported. As before, bisection using Score Difference accurately identified the target number of iterations with an estimated ratio of 1.01 and 1.00 in the two trials. In contrast with previous results, True Skill produces similarly accurate results.

V. Searching for the Initial Boundaries

The bisection procedure assumes that it is possible to provide a minimum and maximum number of iterations to bound the search. It also assumes that the estimated value of MCTS complexity of the target player lies in such an interval. In the previous experiments, we provided the boundaries to guarantee such constraints. In the second set of experiments, we introduced a procedure to compute the initial boundaries and applied it to target players implemented using MCTS with a known number of iterations. Then, we applied bisection search to evaluate the MCTS complexity of the target algorithms using such boundaries.



(a) (b)

Figure 2: Match quality as a function of the tying and winning probabilities using (a) Score Difference and (b) True Skill.

			Score Difference			True Skill		
Target	*min*	*max*	Lower	Upper	Estimated	Lower	Upper	Estimated
1000	1	5000	0.93±0.04	1.11±0.04	1.01±0.02	0.68±0.09	1.27±0.14	0.98±0.03
2000	1	5000	0.97±0.02	1.03±0.03	1.00±0.02	0.90±0.05	1.14±0.03	1.02±0.02

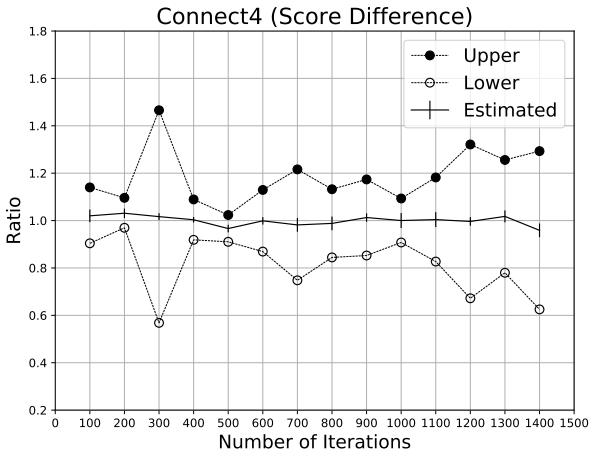
Table I: Bisection search applied to evaluate MCTS players on 6×6 Othello: (a) Score Difference and (b) True Skill; values are averages over ten runs.

Algorithm 3 shows the procedure to compute the initial lower and upper bounds for the bisection search. It takes as input the target player (*target*), an MCTS player (*mcts*), the initial bounds (*min* and *max*), and the number of matches used for comparing the players. At first, the algorithm initializes the bounds and sets the termination criterion to false (lines 2-4). Then, it compares the players over n matches (line 7); if the MCTS player wins more matches (line 8), the process ends and the current bounds are returned. Otherwise, the bounds are updated by setting the lower bound to the current upper bound (line 11) and doubling the current upper bound (line 12). When updating the bounds, we introduce a random perturbation to increase variation to the search process and thus to the computed bounds. To obtain a robust evaluation of the bounds, in the experiments presented in this paper, we repeated Algorithm 3 ten times and computed the final bounds as the average over the ten runs. Algorithm 3 increases the upper bound exponentially and tends to produce wide intervals when facing competitive players that might lose only to MCTS players with a high number iterations. Indeed the procedure might be improved by updating the upper bound with increases adapted to the MCTS losing rate (the lower it is, the smaller the increases of the number of iterations).

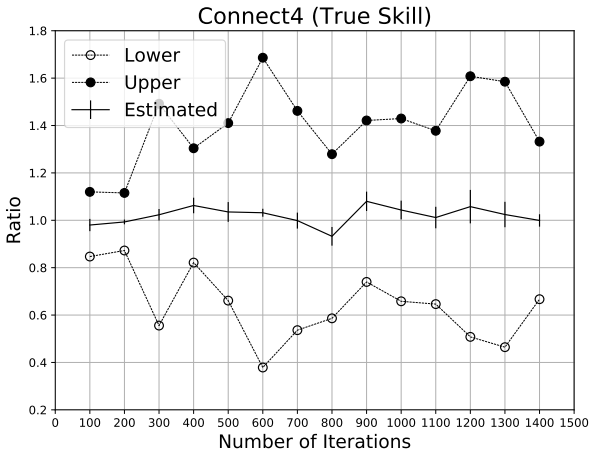
Algorithm 3 Compute the bounds for bisection search

```
1:  function ComputeBounds(mcts, target, min, max, n)
2:      upper ← max
3:      lower ← min
4:      end ← false
5:      repeat
6:          mcts.iterations ← upper
7:          stats ← PlayGame(mcts, target, n)
8:          if stats.wins(mcts)≥ stats.wins(target) then
9:              end ← true
10:         else
11:             lower ← upper - Random(0.1×upper)
12:             upper ← 2×upper + Random(0.1×upper)
13:         end if
14:     until not end
15:     return lower, upper
16: end function
```

We applied bisection search to evaluate MCTS players with 500, 1000, and 2000 iterations with Connect4, Awari, and Othello on a 6×6 board. Initially, for each game, we run Algorithm 3 ten times and took the average lower and upper bound returned by the algorithm as the initial bounds for the subsequent bisection procedure. For this purpose,

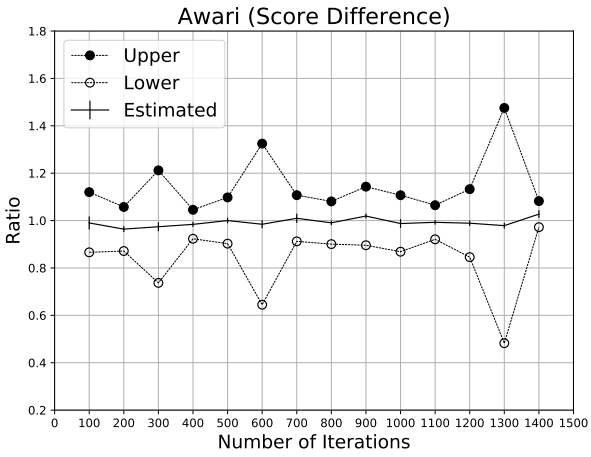


(a)

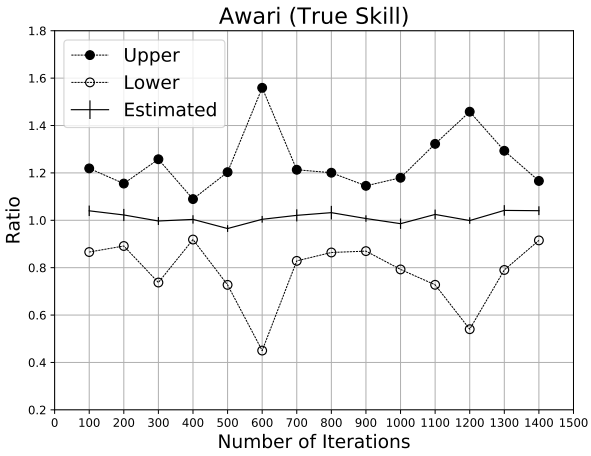


(b)

Figure 3: Bisection search applied to evaluate MCTS players using different number of iterations on Connect4: (a) Score Difference; (b) True Skill; values are averages over ten runs.



(a)



(b)

Figure 4: Bisection search applied to evaluate MCTS players using different number of iterations on Awari: (a) Score Difference; (b) True Skill; values are averages over ten runs.

ComputeBounds started with an initial lower bound (min in Algorithm 3) of 1, an initial upper bound (min in Algorithm 3) of 100, and n of 100. Next, we applied the bisection procedure using the same parameters used in the first set of experiments for the boundary values computed using the above procedure. Table II reports the results of the experiments using the mean and standard error. Column *Target* reports the number of iterations; *min* the computed lower bound; *max* the computed upper bound; for Score Difference and True Skill, it also reports the average lower and upper bounds returned by the bisection procedure as well as the average estimated MCTS complexity (columns *Lower*, *Upper*, *Estimated*); all values are reported as the ratio with respect to the target number of iterations. Both approaches accurately evaluate the target players reporting values near to one. In few cases, bisection underestimates or overestimates the target player reporting like for instance for Othello 6×6 for 1000 and 2000 iterations using Score Difference (with values of 0.96 ± 0.02 and 0.97 ± 0.03) and True Skill (with a value of 1.08 ± 0.06). The analysis of the intervals generated by the bisection procedure shows that overstimation and underestimation typically happened when the average of the current bounds generated a value of iterations near the target value but the statistics of the evaluation caused the update toward the wrong side of the range making the target value unreachable. However, note that the bisection guided the search toward the correct value so that the final ratio is still quite near to one.

VI. Evaluation of Artificial Players

Finally, we applied our approach to evaluate players implementing different strategies that were available online. We used the same setup of the previous experiments and applied the procedure to compute the search boundaries (Algorithm

Game	Target	*min*	*max*	Score Difference			True Skill		
				Lower	Upper	Estimated	Lower	Upper	Estimated
Connect4	500	395±21	839±44	0.93±0.04	1.11±0.04	1.01±0.02	0.68±0.09	1.27±0.14	0.98±0.03
Connect4	1000	640±71	1357±71	0.97±0.02	1.03±0.03	1.00±0.02	0.90±0.05	1.14±0.03	1.02±0.02
Connect4	2000	1297±146	2792±312	0.86±0.03	1.12±0.07	0.98±0.03	0.78±0.02	1.16±0.04	0.97±0.02
Awari	500	398±21	850±44	0.97±0.02	1.04±0.02	1.01±0.01	0.87±0.03	1.15±0.04	1.01±0.02
Awari	1000	769±56	1657±122	0.95±0.04	1.13±0.05	1.04±0.01	0.95±0.04	1.13±0.04	1.04±0.03
Awari	2000	1427±154	2994±321	0.93±0.01	1.05±0.03	0.99±0.01	0.79±0.04	1.31±0.07	1.05±0.02
Othello 6×6	500	362±27	801±60	0.99±0.01	1.07±0.04	1.03±0.02	0.87±0.03	1.16±0.08	1.02±0.03
Othello 6×6	1000	722±66	1567±139	0.86±0.05	1.06±0.05	0.96±0.02	0.91±0.06	1.13±0.03	1.02±0.02
Othello 6×6	2000	1518±149	3226±311	0.88±0.04	1.06±0.04	0.97±0.03	0.86±0.06	1.29±0.18	1.08±0.06

Table II: Bisection search applied to evaluate MCTS players on Connect4, Awari, and Othello 6×6 starting from boundaries computed using Algorithm 3.

Game	Target	*min*	*max*	Score Difference			True Skill		
				Lower	Upper	Estimated	Lower	Upper	Estimated
Connect4	Random	1.00	105±35	1.83±0.48	3.83±0.70	2.33±0.56	2.50±0.50	4.00±0.58	2.80±0.49
Connect4	NFriendsFloyd	1.00	105±35	20.83±0.31	23.33±0.76	21.67±0.33	21.00±0.60	22.00±0.60	21.00±0.60
Connect4	MiniMax	1.00	105±35	36.00±11.17	71.50±10.71	53.50±1.61	57.80±3.66	59.40±3.56	58.20±3.57
Awari	Random	1.00	105.10±35.14	1.33±0.21	2.67±0.33	1.50±0.22	1.10±0.10	3.70±0.62	2.00±0.37
Awari	Divilly	1.00	103.20±34.49	37.00±8.12	60.33±10.14	48.17±2.55	30.80±5.51	62.30±8.28	46.30±2.38
Awari	MiniMax	194.60±72.18	417.60±156.03	267.17±12.75	289.83±9.01	278.17±10.20	157.30±19.37	418.90±33.43	288.00±10.47
Othello 6×6	Random	1.00	105.00±35.12	1.17±0.17	2.50±0.34	1.33±0.21	1.10±0.10	3.30±0.30	1.70±0.15
Othello 6×6	Simple	1.00	104.40±34.95	25.00±5.03	34.83±4.30	29.50±1.52	9.10±4.12	52.20±1.87	30.50±1.41

Table III: Bisection search applied to evaluate players implementing different strategies that were available online.

3) first, and the bisection search next (Algorithm 1). MCTS player parameters were set as in the previous experiments. For Connect4 we evaluated the basic random player, a player implementing an intermediate strategy (NFriendsFloyd[3]), and a player implementing an alpha-beta pruning using a heuristic to evaluate partial boards.[4] For Awari, we evaluated the basic random player; the greedy player discussed by Divilly and O'Riordan [10] which implements a weighted combination of several known heuristics; an alpha-beta pruning search using a heuristic to evaluate intermediate states.[5] For Othello, we evaluated the basic random player and a player using a value function to decide the best next move.

Table III reports the results of the experiments as the mean over ten runs and the corresponding standard error. In this case, we report the raw values and not the ratio since we don't have a reference value of iterations as in the previous experiments. As can be noted, Score Difference and True Skill return basically the same values. In all the evaluations of random players, the bisection process, both with Score Difference and True Skill, returned on average 1-3 iterations that is coherent with the workings of MCTS: with just a couple of iterations, the selection of the action to perform is basically random.

In Connect4, bisection produced an increasing value of MCTS complexity as the strength of the strategy increased. The random strategy corresponds to a value of MCTS complexity around 2-3 iterations; for the NFriendsFloyd strategy, which is advertised as an intermediate strategy, bisection computed an MCTS complexity of around 21 iterations; finally, for the alpha-beta pruning player, bisection returned a complexity of around 53-58 iterations. Also in Awari, bisection produced an increasing value of MCTS complexity coherently with the strength of the players. It is around 2 for the random player, around 46-48 for the greedy player developed by Divilly and O'Riordan [10] which combines six different basic playing strategies, while for the alpha-beta pruning player, the returned MCTS complexity is around 288. Similar results were produced for Othello 6×6. Bisection evaluates the MCTS complexity of the random player around 2 while, for the player using a simple value function, bisection returns an MCTS complexity of around 30.

It is worth noting that by using iterations to evaluate the strength of players, we can somehow compare algorithms that are incomparable in practice. For example, Table III shows that the alpha-beta pruning for Connect4 has a lower complexity than the alpha-beta pruning for Awari. In fact, the former has an MCTS complexity of around 21, the latter of around 46-48.

VII. Conclusions

We introduced the concept of MCTS complexity of an artificial player as the minimum number of iterations that a vanilla MCTS needs to perform equivalently to the target player. We introduced a bisection procedure to compute such value and two criteria to check when two players perform similarly, namely, Score Difference and True Skill. The former provides a hard threshold to stop bisection when the difference in the number of wins is below a given threshold. The latter computes the match quality as an estimate of the draw probability of the match. We evaluated our approach first by computing the complexity of players implemented using MCTS with a known number of iterations. Next, we applied our approach to compute the MCTS complexity of players implementing different strategies.

[3]https://archive.codeplex.com/?p=connect4

[4]http://users.softlab.ntua.gr/~ttsiod/score4.html

[5]https://github.com/neelamgehlot/Mancala-Game-Playing-Agent

The results we presented show that bisection can accurately identify the number of iterations used in target MCTS players. They also show that, when applied to players implementing unknown strategies, our approach produces results that are coherent with the underlying complexity of the evaluated strategies assigning higher values of MCTS complexity to stronger players. Noticeably, by using MCTS iterations to evaluate the strength of players, we may compare the complexity of algorithms that would be otherwise incomparable (if not from a pure algorithmic complexity viewpoint) like for instance a heuristic for Connect4 and alpha-beta pruning for Awari.

Our approach is limited in several respects. Firstly, it is computationally very expensive to derive reliable evaluations. Although bisection partitions the initial range of iteration values logarithmically, each evaluation requires a substantial number of matches. Note however that the use of GPU might help speed up the process in this respect. Secondly, MCTS-complexity values depends on the parameters of the MCTS used in the bisection (our touchstone) which also depends on the reward function used by the games. For instance, if we were to repeat the same experiments using an optimized version of MCTS (e.g. using RAVE [12]) or a different reward function we would probably get different values, although the ranking of the target algorithms would be probably the same.

Finally, our approach might be extended in principle to evaluate the MCTS complexity of a game instead of a player. In this case, we define the MCTS-complexity of a target game as the minimum number of iterations m^* such that, any other MCTS player using a higher number of iterations m would perform equivalently to the MCTS player using m^* iterations. The procedure to compute m^* works similarly to the procedure to determine the upper bound for the bisection search.

REFERENCES

[1] Tom and and Minka. Trueskill(tm): A bayesian skill rating system. pages 569–576. MIT Press, January 2007.

[2] Broderick Arneson, Ryan B. Hayward, and Philip Henderson. Monte Carlo Tree Search in Hex. *IEEE Trans. Comput. Intellig. and AI in Games*, 2(4):251–258, 2010.

[3] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002. cited By 1266.

[4] Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(1):1–43, 2012.

[5] Alex J. Champanard, Tim Gosling, and Piotr Andruszkiewicz. Monte-carlo tree search in total war: Rome ii's campaign ai, 2014.

[6] Computer Go Group at the University of Alberta. Fuego. http://fuego.sourceforge.net/.

[7] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers, editors, *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2006.

[8] Peter I. Cowling, Edward Jack Powley, and Daniel Whitehouse. Information set monte carlo tree search. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(2):120–143, 2012.

[9] Peter I. Cowling, Colin D. Ward, and Edward Jack Powley. Ensemble determinization in monte carlo tree search for the imperfect information card game magic: The gathering. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(4):241–257, 2012.

[10] Colin Divilly, Colm O'Riordan, and Seamus Hill. Exploration and analysis of the evolution of strategies for mancala variants. In *2013 IEEE Conference on Computational Inteligence in Games (CIG), Niagara Falls, ON, Canada, August 11-13, 2013*, pages 1–7. IEEE, 2013.

[11] Frederik Frydenberg, Kasper R. Andersen, Sebastian Risi, and Julian Togelius. Investigating MCTS modifications in general video game playing. In *2015 IEEE Conference on Computational Intelligence and Games, CIG 2015, Tainan, Taiwan, August 31 - September 2, 2015*, pages 107–113, 2015.

[12] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artif. Intell.*, 175(11):1856–1875, 2011.

[13] Ahmed Khalifa, Aaron Isaksen, Julian Togelius, and Andy Nealen. Modifying MCTS for human-like general video game playing. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 2514–2520. IJCAI/AAAI Press, 2016.

[14] Levente Kocsis and Csaba Szepesvári. *Bandit Based Monte-Carlo Planning*, pages 282–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[15] Richard E. Korf and Larry A. Taylor. Finding optimal solutions to the twenty-four puzzle. In William J. Clancey and Daniel S. Weld, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 2.*, pages 1202–1207. AAAI Press / The MIT Press, 1996.

[16] HJ Van den Herik LV Allis, M Van der Meulen. Databases in awari. *Heuristic Programming in Artificial Intelligence*, 1991.

[17] Jeff Moser. Computing your skill. http://www.moserware.com/2010/03/computing-your-skill.html, mar 2010. Code available at https://github.com/moserware/Skills.

[18] Gwaredd Mountain. Tactical planning and real-time mcts in fable legends, 2015.

[19] Mark J. Nelson. Investigating vanilla MCTS scaling on the GVG-AI game corpus. In *IEEE Conference on Computational Intelligence and Games, CIG 2016, Santorini, Greece, September 20-23, 2016*, pages 1–7. IEEE, 2016.

[20] JAM Nijssen and Mark HM Winands. Monte-Carlo Tree Search for the Game of Scotland Yard. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 158–165. IEEE, 2011.

[21] Stefano Di Palma and Pier Luca Lanzi. Traditional wisdom and monte carlo tree search face-to-face in the card game scopone. *IEEE Trans. Games*, 10(3):317–332, 2018.

[22] Marc JV Ponsen, Geert Gerritsen, and Guillaume Chaslot. Integrating opponent models with monte-carlo tree search in poker. In *Interactive Decision Theory and Game Theory*, 2010.

[23] Larry Russ. *The complete mancala games book*. publishers group west, 2000.

[24] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484 EP –, 01 2016.

[25] Istvan Szita, Guillaume Chaslot, and Pieter Spronck. Monte-carlo tree search in settlers of catan. In H. Jaap van den Herik and Pieter Spronck, editors, *Advances in Computer Games, 12th International Conference, ACG 2009, Pamplona, Spain, May 11-13, 2009. Revised Papers*, volume 6048 of *Lecture Notes in Computer Science*, pages 21–32. Springer, 2009.

[26] Joseph Walton-Rivers, Piers R. Williams, Richard Bartle, Diego Perez Liebana, and Simon M. Lucas. Evaluating and modelling hanabi-playing agents. *CoRR*, abs/1704.07069, 2017.