Analysing KataGo: A Comparative Evaluation Against Perfect Play in the Game of Go

Asmaul Husna(✉) and Martin Müller

University of Alberta, Edmonton, Canada
{asmaul,mmueller}@ualberta.ca

Abstract. Much of the research on board games focuses on strong play and on solving games exactly. Recently, programs such as AlphaZero have reached superhuman level in board games such as chess and Go. We study the gap between such AI systems and perfect play, in order to deepen our understanding of their current strengths and limitations. Our study uses Go endgame puzzles with special combinatorial sum game structure, for which an optimal solver is available. We develop an extended Go endgame dataset labelled with exact scores and optimal moves. We evaluate KataGo, the strongest open source AlphaZero-derived program for the game of Go, on these puzzles. We study how the training of different neural networks and the amount of search used affect KataGo's ability to play perfectly. We observe improved move selection with strong policies, and measure the effect of different MCTS search settings, as well as the challenges KataGo faces in competing against an exact solver. We further analyse move choices in terms of changes of average action value, lower confidence bound, winrate, and number of visited nodes in the MCTS search of KataGo. On our perfect game dataset, KataGo achieves a 90.8% success rate in matches against the exact solver.

Keywords: Performance Evaluation · Safety-Critical Systems · AlphaZero · KataGo · Exact Solver

1 Introduction

DeepMind's AlphaZero (AZ) program [12] has greatly advanced the state of the art in playing two player board games. It convincingly beat top human players and previous programs by learning from scratch. Search methods inspired by AZ have been applied to many problems beyond games [4]. There is much interest in further applications in safety-critical systems, such as autonomous driving, where even a tiny mistake can cause big problems, and having accurate solutions is extremely important. Autonomous driving includes a wide range of tasks, including lane keeping [5], overtaking [6], and making higher-level driving decisions [3].

Can AlphaZero-based algorithms find exact solutions? We explore KataGo, a strong AlphaZero-derived open source Go program [14]. We test how well

M. Hartisch et al. (Eds.): CG 2024, LNCS 15550, pp. 43–53, 2025.
https://doi.org/10.1007/978-3-031-86585-5_4

this program plays Go endgame puzzles compared to perfect play. We aim to better understand the learning process of KataGo and its limits. We study its behavior in several scenarios: when using either a strong or a weak neural network, and with different amounts of search as well as without search. We use endgame puzzles from the literature [1,7], and create a larger dataset that is labeled with perfect solutions by using an exact endgame solver [8]. We design a methodology with experiments and analysis in order to investigate the following research questions:

- What are the differences in move selection between stronger and weaker neural networks?
- How does the addition of a small MCTS search enhance move selection compared to using only a neural network?
- What is the impact of increasing the search budget on the overall performance?
- How good is KataGo at finding a best incentive move in the sense of combinatorial game theory, compared to "just" finding a minimax optimal move?
- How often can KataGo win games against an exact solver in matches from endgame starting positions?
- Are there any cases where using deeper search adversely affects move selection compared to using a small search?

2 Related Work

Comparing a heuristic search-based algorithm against a perfect solution offers important insights into its quality. In related games research, Haque et al. [2] compare Leela Chess Zero (lc0), an open-source chess program derived from AlphaZero, against perfect chess endgame play. They find that more training and deeper search are both very effective for increasing the number of optimal moves found. However, they also identify some cases where a neural network suggests the correct move, but a small search using MCTS (Monte Carlo Tree Search) switches to the wrong move. They also find that some difficult 5-piece endgames are far beyond the current abilities of even the strongest available network combined with deep search. Sadmine et al. [11] extend this study by comparing lc0 and Stockfish, another strong open source chess engine which uses a very different neural network architecture. They use an Average Centipawn Loss (ACPL) measure to quantify how much value is lost by wrong moves, and focus on difficult situations where errors are more common, which include imbalanced positions with one extra black pawn against a stronger extra white piece.

Romein and Bal [10] analyse the game of Awari using a perfect-play database. They find that the top programs Marvin and Softwari, which were previously assumed to be almost perfect, played a correct move in only 82% and 87% of the cases, respectively.

Wang et al. [13] train adversarial policies which can exploit specific weaknesses in KataGo and win over 99% of games against KataGo using no search, and more than 97% against KataGo with some search.

3 Background

3.1 Solving Go Endgame Puzzles

Solving a game means finding an optimal strategy that guarantees the game-theoretical best outcome, such as a win or a draw for a player, no matter how the opponent plays. In two-player games of perfect information, all of the game state is known to both players. However, many game positions are too complex to be solved in practice. In Go endgame puzzles, most of the board has already been secured by one of the players, and they focus on small but crucial battles on the rest of the board, in order to increase their territory and reduce their opponent's. Figure 1 shows the safe stones and territories of a Go endgame position.

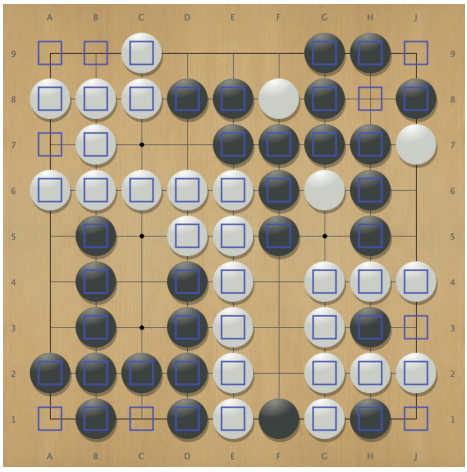


Fig. 1. Safe stones and territories of a Go endgame position.

3.2 Exact Solver for Go

An exact solver for Go can find the best moves and outcomes for a given Go position. Due to the game's complexity and large number of possible moves, such solvers only exist for special cases: small boards, and endgame puzzles with extra structure such as the one in Fig. 1. In our work, we use a solver [7] that can handle endgame puzzles with solution lengths exceeding 60 moves. It uses a technique called decomposition search [8] to find the outcome and the best

moves. The solver first finds safe stones and territories, then identifies independent local subgames, and performs local combinatorial game search (LCGS) in each subgame. Next, the algorithm evaluates the combinatorial game value for each subgame, and tries to find a move with the highest incentive overall. **Incentive** is a combinatorial game concept that measures the improvement to a game position made by a move. Incentives are only partially ordered. If one incentive dominates all others, as often happens in Go endgames, then a move with this incentive is optimal and can be played without further search. If several incentives are incomparable in the partial order, then an optimised **minimax search** among the corresponding candidate moves is used to find their minimax score, and a minimax-optimal move. In our experiment, we study both cases: positions where a unique **best incentive** exists, and those with two or more nondominated incentives. We study minimax-optimal play, as well as whether a policy network can identify best incentive moves and separate them from locally weaker ones.

3.3 KataGo

KataGo [14] is an open-source Go program that implements and extends the AlphaZero algorithm. It learns from scratch through self-play. Unlike AlphaZero, it includes domain-specific features to boost learning. Its key contributions include general purpose improvements over AlphaZero, such as better data balance, focused training, and enhanced neural networks with global pooling layers. Go-specific improvements include extra input features to improve learning, and helpful auxiliary targets such as as predicting point ownership and the final score.

KataGo is the strongest open-source Go program, and is very widely used as a study tool in the Go community, by amateurs and professionals alike. Therefore a study of its limitations is directly relevant for this audience.

4 Dataset and Evaluation Method

4.1 Original and Modified Dataset

We used the 22 endgame problems labeled $C.1, C.2, ..., C.22$ from Berlekamp and Wolfe's "Mathematical Go: Chilling Gets the Last Point" [1]. We refer to these as the **original problems**. We also used the **modified problems** from [8]. These modified versions are equivalent in terms of local endgame values, but clearly separate endgame areas by safe stones and territories. This allows the exact solver to analyse each small area separately, and solve the overall problem. In each problem, white is to play, and if both players play perfectly, then white wins by 0.5 points.

4.2 Extended Datasets

We use the exact solver to generate an extended dataset by playing perfect games starting from different states in the modified endgame problems. As each

game progresses, we adjust the komi when stones are captured. We then play matches between the exact solver and KataGo from these starting positions. If KataGo suggests a move that is not winning as indicated by the solver, then we verify that it leads to KataGo losing the game. We expand our dataset for each endgame position as follows:

- Find the winning move of that board position using the exact solver
- Play the winning move on the board and get the next starting position
- Adjust the komi if stones were captured
- Repeat until the end of the game

To expand the dataset, we use the modified problems $C.1, C.2, C.3, C.6, C.7$, $C.8, C.9, C.10$, and $C.21$. We name this collection of data **perfect games**. To add more 19×19 endgames to our dataset, we use an extension of the subsets of $C.11$ from [9]. From these subsets, we generate additional data and name these endgames **C.11 subsets**. All these endgame problems are stored in Smart Game Format (SGF), a popular file format.

4.3 Splitting the Dataset by Existence of a Dominating Incentive

We divide our dataset of perfect games and C.11 subsets into two categories to evaluate KataGo's decision-making ability. For positions in category **DI**, a single dominating incentive exists. The **no-DI** positions do not have a single dominating incentive. Among the set of moves with two or more non-dominating incentives, the best ones must be determined by minimax search.

4.4 Engine Settings

We used KataGo version v1.12.4 to analyse our endgame dataset. We also used two different KataGo neural networks. For the best performance, we chose a strong network called "kata1-b18c384nbt-s5832081920-d3223508649" with 18 blocks and 384 channels. Its self-play Elo rating is 13488.6 ± 14.1. Our weaker option is a smaller network, "kata1-b6c96-s37368064-d5536083" with 6 blocks and 96 channels, and a 7098.1 ± 19.0 elo rating.

For computational resources, we used one CPU core to process datasets and OpenCL GPU to run KataGo. For example, the strong network took 64.1 s to process 126 endgames with 100 search nodes each. The weak network did the same task in 1.2 s. Analysing all perfect games with the strong network and 102,400 search nodes took nearly 5 h.

5 Experiments and Analysis

5.1 Evaluating KataGo with Basic Settings

To evaluate KataGo's performance, we run two types of tests: one uses only a neural network policy, without search, while the other test applies the network

during search as usual. In both types of tests we try the weak and the strong version of the network. In each endgame position, we count a KataGo move as correct if it is one of the known set of optimal moves. We consider both notions of optimality, incentive optimal and minimax optimal. We measure the number and percentage of correct moves among all generated moves. Since KataGo's results can change between runs, we run each experiment five times and average the results. Table 1 summarises the average performance of KataGo for each of the four test sets, in four scenarios: using the weak and the strong policy without search, and using both policies with a search with a limit of 100 node visits.

Table 1. Total number of correct moves along with average success rate by the weak and strong policy, and policies with 100 visits search.

Dataset	Test Cases	Average Total Number of Correct Moves and Success Rate (%)			
		Policy		Search, 100 visits	
		Weak	Strong	Weak	Strong
Original	22	8.8(40%)	12.8(58.2%)	9.6(43.6%)	16.2(73.6%)
Modified	22	7.8(35.5%)	13.6(61.8%)	9(40.9%)	16.8(77.3%)
Perfect games	126	98.8(78.4%)	118.8(94.3%)	105.8(84%)	123.8(98.3%)
C.11 subsets	371	355.6(95.8%)	369.6(99.6%)	361.4(97.4%)	370(99.7%)

As expected, in general the strong policy outperforms the weak one in all settings and across all datasets. Among datasets, the success is higher in perfect games and C.11 subsets, likely because many of these endgames offer more winning moves (4 to 10) compared to the challenging other sets with only one or two correct moves. Furthermore, when playing out perfect games, the endgame size is gradually reduced, with fewer contested points, often leading to progressively easier to solve positions. The results also show that adding a small search to the weak net does not improve results much for the challenging original and modified problems. Many of these problems remain out of reach. In contrast, the more focused searches using the strong network shows more improvement.

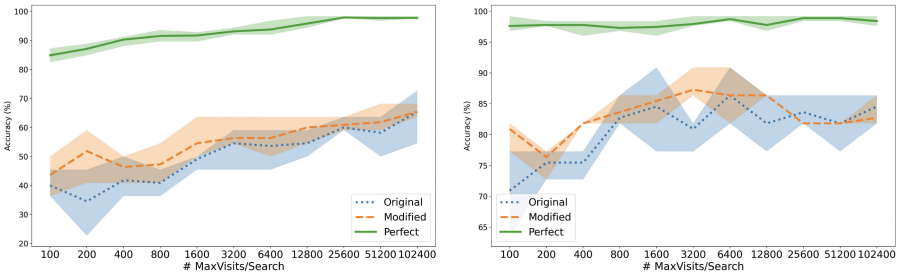
Scaling the Search. We increase the search amount by ten successive doublings from 100 to 102,400 maximum visits for all test cases. Figure 2 shows the average success rate while using the weak and strong policies in these settings. In Fig. 2 (a), the success rate of the weak policy steadily rises for original, modified, and perfect games. However, this combination still struggles with original and modified problems, reaching only about 60% accuracy with 102,400 visit searches. The scaling results for the strong policy in Fig. 2 (b) are surprising. Initially, the success rate increases, but then it starts to deteriorate. This suggests that in several endgame scenarios, picking a correct move is more due to chance, even at this level of search. For C.11 subsets, we focus on one endgame unsolved by the strong policy with 100 visits in Table 1. It still cannot be solved by either policy with any amount of search that we tried.

5.2 Evaluating KataGo's Incentive- and Minimax-optimal Play

In Subsect. 4.3 we split our datasets into DI, the positions where a dominating incentive exists, and no-DI, where it does not. In all cases, we evaluate whether KataGo's move is minimax optimal. For test set DI, we additionally check if KataGo's move is incentive-optimal. In each position, the set of incentive-optimal moves is a subset of the minimax-optimal ones. The sets can be equal, and always contain at least one move. Table 2 shows the average results for DI with weak and strong policies, and for the same policies with a 100 node search. We use the datasets perfect games and C.11 subsets, because for these sets, both incentive and minimax-optimal moves are available from the exact solver. From the results of Table 2, we find that KataGo struggles more to identify best incentive moves compared to minimax ones. This shows in the success rates of finding incentive and minimax optimal moves. We see a large difference, ranging from 10 to 40 percent. It appears that KataGo focuses more on finding any win, rather than relying on the policy learning the small differences in move strength. We only show the results for the DI test set here, as our no-DI set is too small to analyze the results in a meaningful way.

Table 2. Total number of correct moves and average success rate for weak and strong policy, and with 100-visit search, for the DI test set.

Name of Dataset	Number of Endgames	Total Number of Correct Moves with Avg Success Rate (%)			
		Weak policy		Strong Policy	
		Incentive optimal	Minimax optimal	Incentive optimal	Minimax Optimal
Perfect games	125	70.8(56.6%)	96.8(77.8%)	103(82.4%)	115.8(92.6%)
C.11 subsets	345	190(55.1%)	328.8(95.3%)	251.6(72.8%)	345(100%)
		Weak policy + 100 visit search		Strong Policy + 100 visit search	
		Incentive optimal	Minimax optimal	Incentive optimal	Minimax Optimal
Perfect games	125	89.2(71.4%)	106.2(85%)	106.2(85%)	122(97.6%)
C.11 subsets	345	235(68.1%)	333.8(96.8%)	254.6(73.8%)	345(100%)



(a) KataGo's weak policy with search. (b) KataGo's strong policy with search.

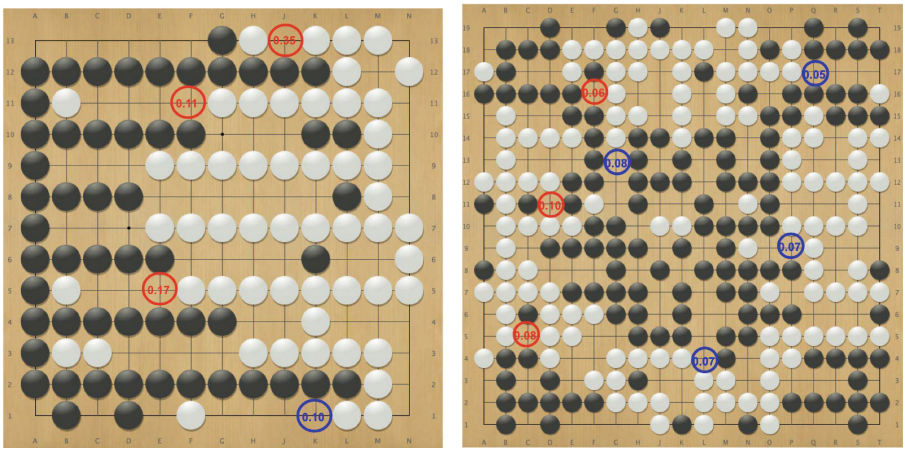
Fig. 2. Average, minimum and maximum success rate of KataGo's weak and strong policy with different amounts of search.

5.3 Playing Matches Between the Exact Solver and KataGo

Matches between the exact solver and KataGo show the importance of perfect play. We use the strong policy with 500 visit search. Since the exact solver can solve every position in the perfect games and C.11 subsets, we use these two sets for our experiment. KataGo preserves the win in 90.8% of the perfect games and wins 100% of the matches starting from the C.11 subsets. In Sect. 5.1, we identified an endgame in the C.11 subsets that KataGo is never able to solve. However, this position doesn't affect the win rate here, as Black (the losing side) is to play here.

5.4 Case Studies of Interesting Mistakes

Figure 2 shows that the number of mistakes fluctuates with larger searches. Several endgames remain unsolved even with our deepest search. For further analysis, we run selected endgames with more than 200k search visits and observe the changes of utility (action value), lower confidence bound (lcb), winrate, and number of visited nodes explored during the search process. Two examples discussed below are shown in Fig. 3.

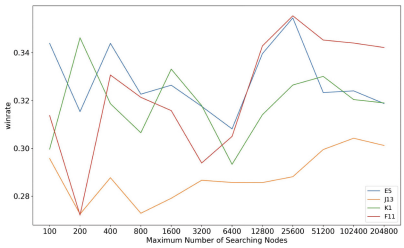


(a) Original problem *C*.3. (b) Modified problem *C*.11.

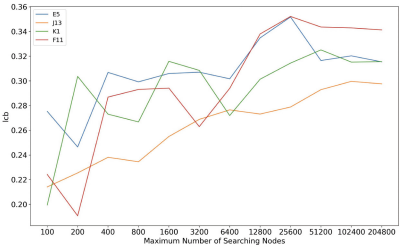
Fig. 3. Examples for two types of interesting mistakes. KataGo's suggested moves are shown as red circles, and winning moves as blue circles. The number inside each circle is the policy probability for that move. (Color figure online)

The original problem *C*.3 in Fig. 3(a) is not solved by KataGo even with deep search. In different searches, KataGo suggests three different moves: *J*13, *E*5, and *F*11, while the only winning move is *K*1. Figure 4(a-d), shows the changes of winrate, lcb, utility and the number of visited nodes during the search. *J*13 has high prior probability in Fig. 3(a), but as the search progresses, it drops. Move

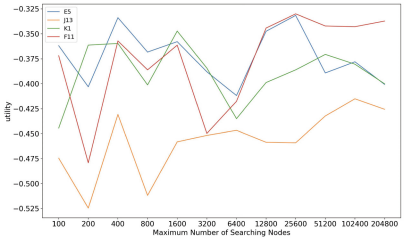
$F11$ starts out with a low prior, but with more search it becomes KataGo's preferred move. The optimal move $K1$ goes up, then down again. The win rate



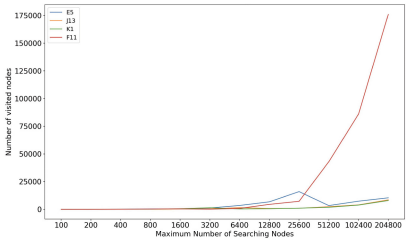
(a) Winrate.



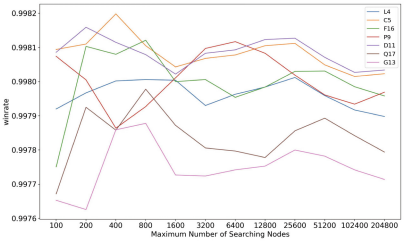
(b) Lower confidence bound (lcb).



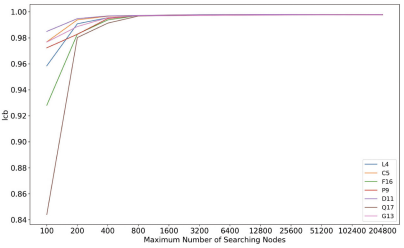
(c) Utility (action value).



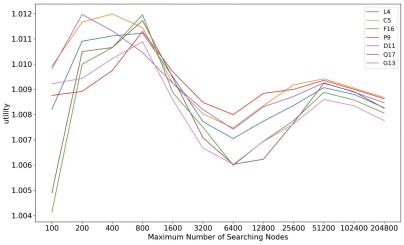
(d) Number of visited nodes.



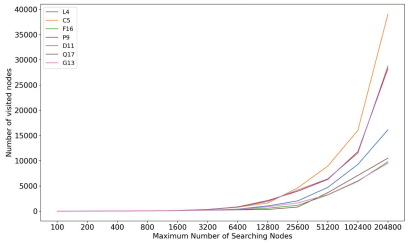
(e) Winrate.



(f) Lower confidence bound (lcb).



(g) Utility (action value).



(h) Number of visited nodes.

Fig. 4. The changes of winrate, lcb, utility, and number of visited node during the search.

and lcb of all moves in Fig. 4(a and b), remain below 0.4. KataGo completely fails to recognize that $K1$ wins. Figure 4(d) shows that after 25,600 visits, almost all visits go to $F11$, a losing move.

Small Search Correct but Longer Search Wrong. In modified problem $C.11$ shown in Fig. 3(b), KataGo correctly identifies winning moves with a small search but switches to a wrong move after a longer search. The winning moves of this position are $G13$, $L4$, $P9$, and $Q17$. However, KataGo also suggests the losing moves $D11$, $F16$, and $C5$ in different searches.

The changes of winrate, lcb, utility and the number of node visits in Fig. 4(e-h) show that initially, the winning moves are visited less often. After 1600 visits, there is a significant improvement for $P9$, and KataGo starts selecting this winning move. However, with deeper search the value of $P9$ decreases again, and the losing move $C5$ becomes KataGo's preferred move. The main issue here is that KataGo cannot distinguish at all between winning and losing moves. The win rate, utility, and lcb of all losing moves in Fig. 4(e-g) remain high. In Fig. 4 (h), KataGo explores $C5$ more in larger searches.

6 Limitations and Future Work

We explored some key questions of move selection, the role of search, and the performance across several Go endgame datasets. The difference between weak and strong neural networks was large. Adding even a small amount of search greatly improved both. In matches against an exact solver, KataGo did well but made mistakes in some simple endgames, particularly those with long corridors of similar values. The network has not learned the correct relative values of those moves, and often prefers a simple capture of slightly lesser value. Larger searches generally boosted KataGo's performance but did not solve all these puzzles.

Limitations. We do not have a large enough set of complex endgames, such as no-DI. Our perfect game and C.11 subsets are relatively much easier to solve than the original and modified Berlekamp/Wolfe problems. We did not run detailed enough experiments to fully understand why KataGo struggles to find the best incentive moves.

Future Work. One interesting future topic is to measure KataGo's winrate errors separately for winning and losing moves. In theory, all winning moves should approach a winrate of 1, and all losing moves should be close to 0. A similar winrate for both winning and losing moves indicates a poor understanding of the position. We can evaluate problem difficulty more clearly by looking at additional factors such as number of winning moves. We could also add more challenging 19×19 test sets by following a narrow winning line from C.11 instead of just removing subgames as in prior work, which tends to make the problems much easier.

References

1. Berlekamp, E., Wolfe, D.: Mathematical Go: Chilling Gets the Last Point. CRC Press (1994)
2. Haque, R., Wei, T.H., Müller, M.: On the road to perfection? Evaluating Leela Chess Zero against endgame tablebases. In: Advances in Computer Games, pp. 142–152. Springer (2022). https://doi.org/10.1007/978-3-031-11488-5_13
3. Hoel, C.J., Driggs-Campbell, K., Wolff, K., Laine, L., Kochenderfer, M.J.: Combining planning and deep reinforcement learning in tactical decision making for autonomous driving. IEEE Trans. Intell. Veh. **5**(2), 294–305 (2019)
4. Kemmerling, M., Lütticke, D., Schmitt, R.H.: Beyond games: a systematic review of neural Monte Carlo tree search applications. Appl. Intell. **54**(1), 1020–1046 (2024)
5. Kővári, B., Hegedüs, F., Bécsi, T.: Design of a reinforcement learning-based lane keeping planning agent for automated vehicles. Appl. Sci. **10**(20), 7171 (2020)
6. Mo, S., Pei, X., Wu, C.: Safe reinforcement learning for autonomous vehicle using Monte Carlo tree search. IEEE Trans. Intell. Transp. Syst. **23**(7), 6766–6773 (2021)
7. Müller, M.: Computer Go as a sum of local games: an application of combinatorial game theory. Ph.D. thesis, ETH Zurich (1995)
8. Müller, M.: Decomposition search: a combinatorial games approach to game tree search, with applications to solving Go endgames. In: IJCAI, pp. 578–583 (1999)
9. Müller, M.: Not like other games - why tree search in Go is different. In: Proceedings of Fifth Joint Conference on Information Sciences (JCIS 2000), pp. 974–977 (2000)
10. Romein, J.W., Bal, H.E.: Awari is solved. ICGA J. **25**(3), 162–165 (2002)
11. Sadmine, Q.A., Husna, A., Müller, M.: Stockfish or Leela Chess Zero? A comparison against endgame tablebases. In: Advances in Computer Games, pp. 26–35. Springer (2023). https://doi.org/10.1007/978-3-031-54968-7_3
12. Silver, D., et al.: A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science **362**(6419), 1140–1144 (2018)
13. Wang, T.T., et al.: Adversarial policies beat superhuman Go AIs. In: ICML 23, vol. 1484, pp. 35655 – 35739 (2023)
14. Wu, D.J.: Accelerating self-play learning in Go. In: AAAI-20 Workshop on Reinforcement Learning in Games (2020)