Developing a game AI for Murus Gallicus

Philip Wilson[1] Andrej Savinov[2] Annabella Kadavanich[3]

Abstract: The development of game AIs has been a popular challenge in the last years. One of the best game agents, *AlphaZero*, was developed by DeepMind in 2017 and superseded by *MuZero* in 2019. Both agents are based on algorithms that perfectly learn to play any game within not even a day, given they are fed the game's rules [Da18]. The development of such game AIs does not necessarily require big computation centers like the ones Google has. In this work, we show how to develop and implement a *Murus Gallicus* game AI using mainly GOFAI (Good Old-Fashioned Artificial Intelligence) methods. We start with a comparison between different search tree algorithms, including MiniMax, NegaMax, NegaScout (principal variation search) and show how transposition tables can be used for optimization. Furthermore, we demonstrate the advantages of a dynamic value function and time management while searching for the best move. Lastly, we evaluate the application of Evolutionary Learning (EL), explaining how we trained specific parameters.

Keywords: Artificial Intelligence, Game AI, Evolutionary Learning

1 Motivation

This project was part of a course at the Technische Universität Berlin called "Projekt KI: Symbolische Künstliche Intelligenz"(AI Project: Symbolic Artificial Intelligence). This means that we had no choice in the methods we implemented except for EL. Even though Deep Reinforcement Learning methods are currently the best at playing board games and even simple video games, it does not mean we should forget the old school methods that went out of favor in the 1980s. *MuZero* still gets the same score as a random agent for some Atari games like "Montezumas Revenge"where a combination of reinforcement learning and symbolic representations may be needed to beat these games. In this project, we demonstrate how to implement an AI for the game called *Murus Gallicus* (MG). MG is a board game based on the walls of stone built by the Gauls against Roman aggressors in the Gallic Wars. The game was invented in 2009 by Phil Leduc [Ph19]. We examine the advanced version of MG, which also includes catapults in addition to walls and towers.

[1] TU Berlin, Straße des 17. Juni, 12309 Berlin, Deutschland, philipollaswilson@protonmail.com
[2] TU Berlin, Straße des 17. Juni, 12309 Berlin, Deutschland, andrej.savinov@campus.tu-berlin.de
[3] TU Berlin, Straße des 17. Juni, 12309 Berlin, Deutschland, a.kadavanich@campus.tu-berlin.de

CC BY SA

MG is played on a 7 x 8 board. There are two players, each has 16 stones of their color, where light represents the Romans and dark the Gauls. A tower consists of two stones of the same color stacked on top of each other, while a single stone is called a wall. A catapult consists of three stones of the same color, stacked on top of each other. At the start of the game each player possesses a row of towers. The Romans start the game. On their turn, a player may move a tower, capture with a tower or fire a catapult. Walls cannot be moved. Towers can move to any straight direction, orthogonal or diagonal. Both stones are removed from the starting square and placed, one each, on the next two squares of the chosen direction. Destination squares may not be occupied by adversarial pieces or friendly catapults. If a destination square is occupied by a friendly wall it becomes a tower, and if occupied by a friendly tower it becomes a catapult. Towers can capture an adjacent adversarial wall by sacrificing one of its stones. Both stones are then removed from the board. Towers can downgrade an adjacent adversarial catapult to a tower by sacrificing one of its stones. Two stones are removed in total, one of the attacking tower's stones and one of the defending catapult's stones. Similarly, towers can downgrade a catapult to a wall, by sacrificing two stones in total. Catapults can throw one of their stones two or three spaces away in all five forward directions into an empty or opponent occupied cell. The player who places a wall on the opponents home row wins the game. If a player cannot make a valid move anymore he loses the game.

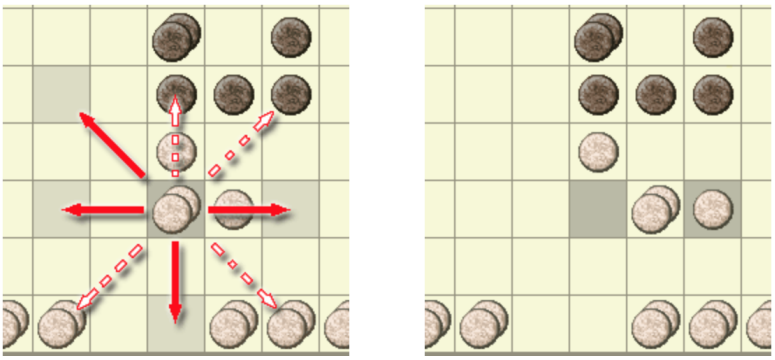

Fig. 1: Exemplary game state (right) showing in which directions a tower can move (left) Source: http://www.iggamecenter.com/info/de/murusgallicus.html, last visited on 15.07.2020

Even though an empirical analysis was not performed, our intuition is that the state and action spaces are smaller than in chess. Unconstrained by the board all possible actions are: a tower can move in eight directions or capture any adjacent enemy wall or downgrade a catapult, and a catapult can fire towards six different locations. Every one of these moves, except the tower move, are guaranteed to reduce the action space, as pieces are either lost or walls are created which have no moves. As there are only seven types of states a square can

have (empty or wall, tower or catapult of either colour) compared to 13 states in chess, and as the board is 7x8 and not 8x8, we also believe the state space to be smaller.

2 Tree search algorithms

This paper is part of an university project at TU Berlin, consisting of different student groups who developed their own game AI. In total seven groups participated. During three milestones (MS), and one pre-milestone, all game AIs competed against each other in a contest simulation environment. During each contest each move had a time limit, therefore the AI that made the smartest cutoffs (using alpha-beta pruning [BM01]) and could search the game tree the deepest, generally won the contest. Therefore, this chapter is dedicated to introducing and comparing different search tree algorithms.

2.1 Minimax

MiniMax is the standard algorithm for two player perfect-information games such as MG. It searches forward to a fixed depth in the game tree, limited by the amount of time available per move. Then, a heuristic evaluation function is applied, which takes a board position and returns a number that indicates how favorable that position is for one player relative to the other. One player, called MAX, seeks to maximize this number, whereas the other player, called MIN, seeks to minimize it. Finally, it recursively computes the values for the interior nodes in the tree according to the maximum rule. The value of a node where it is MAX's turn is the maximum of the values of its children, while the value of the node where MIN is to move is the minimum of the values of its children [Po89].

2.2 Iterative Deepening Negamax

NegaMax [Al90] search is a variant form of the MiniMax search algorithm. We have chosen it in order to simplify the implementation of the MiniMax algorithm. NegaMax takes advantage of the zero-sum property of MG. In other words, the value of a position for MAX is the negation of the value for MIN. Instead of MAX selecting the move with the maximum-valued successor and MIN selecting the move with the minimum-valued successor, they both look for a move that maximizes the negation of the value resulting from the evaluation function.

2.3 NegaScout: Principal Variation Search

NegaScout is a directional search algorithm for computing the MiniMax value of a node in a tree. It is said to be faster than NegaMax with alpha-beta pruning [BM99]. This is true also for our MG AI (see *Fig. 2*). In our implementation we use a null window to reduce the amount of nodes that need to be computed. We used move sorting and only searched the best move with a full window, all other moves were searched using null windows. Though this can be risky, theory shows [Ma86] that if the null window value is bigger than the alpha value and smaller than beta then the move could possibly have a better value and therefore should be searched using a bigger window. If such a situation appeared, another NegaScout computation using a bigger window was calculated.

2.4 Transposition Tables with zobrist hashing

It is possible to arrive at the same game state through different combinations of moves [BUH70]. To avoid calculating the value of the state more than once, the values are stored in *Transposition Tables* (TT) [BTW00]. To increase the search speed during lookups, we also implemented zobrist hashing [JF16]. The depth, at which each value was discovered, is also stored in order to give priority to states whose value was calculated with a larger subtree (the ones with lower depth). When the table fills up, less-used states are removed to make room for new ones.

Fig. 2 shows a comparison of the computed nodes using NegaMax, NegaScout without TT and NegaScout with TT. The x-axis describes the tree depth that was used and the y-axis the amount of nodes that were computed. Taking the average amount of nodes for several different fen strings, we found that NegaScout generates on average almost twice as many nodes as NegaMax and it only takes less than half of the time doing so. Using NegaScout with TT we could decrease our computation time and therefore reach another speed up.

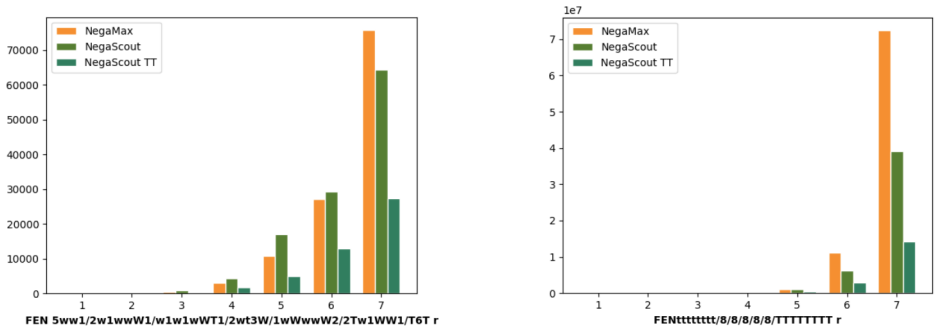

Fig. 2: Comparison of NegaMax, NegaScout without TT & NegaScout with TT for 2 FEN Strings

Comparing the computation time, using NegaScout is 1.5 times faster than NegaMax in average. Furthermore, we noticed that on average NegaScout with TT is 2.9 times faster than NegaMax. Also the direct comparison of NegaScout with TT and NegaScout without TT showed, that **not using TT** almost doubled the computation time. These calculations are based on measuring the time and amount of computed nodes for all three algorithms for approximately 20 different FEN strings and then computing the arithmetic mean.

2.5 Value Function

Besides optimizing the search tree algorithm, our most challenging task while developing the game AI was to design a good value function [RA10]. In our implementation we started with a basic value function, only consisting of a *Piece-Square Table* [CP20] (PST) and then progressively added the following five features:

1) Amount of towers > amount of walls, 2) amount of catapults > amount of walls, 3) mobility 4) Dominance of the middle row and lastly, 5) giving or taking points, depending on whether the amount of our towers is larger than the amount of enemy towers. We adjusted the parameters by adding weights that could easily be changed. With this dynamic value function we were able to win almost all games against the other groups at the second milestone. Furthermore, we had implemented a better defense strategy that forced our AI to build a row of defensive towers in the middle of the board using our PST. Once this position was achieved we deactivated the use of PST and played more aggressively towards the winning line of the opponent. Since each game only consisted of 120s we did not re-activate the PST in case the defense border was destroyed as there was no time to rebuild it. In addition, we implemented a panic mode that got activated once our time limit dropped below a certain threshold. In panic mode the search depth was decreased and a more aggressive play-style was activated.

2.6 Summary

We examined different features in order to develop a good MG AI. We started with very simple features and core functions, e.g. the implementation of a move generator and bitboards in MS1. Our value function at that time was only the PST and we used NegaMax with alpha-beta pruning as our main search tree algorithm. Our time management was static and we would always search depth five without any panic mode. Moving forward to MS2, we implemented NegaScout with a null window, we adapted our static time management to a dynamic one, added a panic mode and variable search depths. We also started implementing TT with Zobrist Hashing which improved our computation time Most important for MS2 was also the change of our value function. Instead of only having two PSTs we added six tables, three for each player and one table for each material (Materials: wall, tower, catapult). In addition, our value function was changed to be a linear combination of seven parameters.

The analysis of the contests in MS1 and MS2 showed that all AIs played without any variations, leading to two adversarial AIs always playing the same game against each other repetitivly. To create a more dynamic value function, some type of learning would be helpful. Therefore, we tried using Machine Learning (ML) techniques of our choice to improve the performance of our AI. We implemented Evolutionary Learning (EL) since there was no MG training data available. In the next chapter we will explain, why we chose this technique, how we trained our AI, which challenges we faced and how our AI performed in the last MS3 contest.

3 Evolutionary Learning

EL [Ri01] is inspired by the evolutionary processes happening in nature. Therefore, every evolutionary algorithm needs a fitness function. A function used to compare organisms in order to decide which ones survive and which ones will pass their genes on to the next generation. As we had no datasets to train this fitness function with, we had to establish the fitness function by means of playing numerous games between the organisms in each generation (SPC) regardless of its impracticality [Da09].

3.1 Parameter Adjustments

The first parameter we had to decide on was the population size. Our initial population size for each epoch was ten. In the following, game AIs are referred to as so called *Individuals*. Furthermore, we decided to only train one parameter at a time. We chose to start with the material values (wall, tower, catapult), since other values were more complex to train (e.g. PSTs). Every Individual in the population had different material values in order to ensure variation. Our contest AI from MS2 was also part of the population. We chose Round-Robin (RR) as the tournament mode. We also considered K.O. tournaments as they finish faster than RR and we hoped to speed up our training.

3.1.1 Crossover

We set the probability for a crossover to 10-15%. When we used these low values the improvement of our population was very slow. Since we only had three weeks of training time, until the next MS contest, we increased the value to 80%. This allowed us to create more variance and speed up our training time.

The crossover was implemented in the following way: First, we determined the current best Individual based on the win rate. This way of selection is called *elitism* [TD94]. We perform a crossover on all Individuals, except our MS2 AI and the selected current best Individual. The new material values were calculated as follows: we randomly chose four Individuals of the whole population and select the best Individual from this group (again

using the win rate). Next, this step was repeated. In this way, we selected two Individuals. This type of selection is called *tournament selection* [Ge20]. It is possible – though rare – that the same Individual is selected twice in this process. If this happens, we randomly re-selected another AI. Next, we created a new Individual object by summing up the material parameter of both Individuals from the tournament selection and dividing this number by two. We did this separately for the wall, the tower and the catapult parameter. This tournament selection process is repeated until a new population of eight Individuals is created (assuming that the best Individual of the population was not our MS2 AI). Then we added the best Individual and our MS2 AI to the new population and therefore had a new population of ten Individuals again, used for training the next epoch.

3.1.2 Mutation

Our mutation process was similar to the crossover (see 3.1.1). The probability was set to 50%. Normally in EL this parameter is very low (< 5%) [Go89] but as we only had limited time and needed to speed up the training process we increased the probability.

Again, we implemented a variation of elitism selection. For each Individual – except the Individual selected by elitism and our MS2 contest AI – we randomly choose the material value for mutation. We then overwrote this value by adding a random ranged offset to the current value of the Individual. This step was repeated until all Individuals mutated.

3.2 Material Parameter

We spent about two weeks setting up our training environment (implementing the classes, controlling our memory overflows, optimizing the parameter and the crossover, see 3.1.1 & 3.1.2) and in parallel, started training the weight parameters that are multiplied with the PST. We noticed that after approximately 20 generations our population would start to **converge** and no further improvements were made. Since this happened continuously for different populations we decided to move on and train other parameters.

In summary, we did find three Individuals that were able to win against our MS2 AI. We also tested these three Individuals in a local contest against the MS2 AI's from the other groups. We had slightly better scores than in MS2. In total, the results were not satisfying and also did not mirror the effort and time we had spent on the training of the parameter.

3.3 Piece-Square Table

We also decided to try training the PST parameter. Instead of just training the material factor of the PST, we trained every index of the PST list. Mutation and crossover rate was the same as in section 3.1.2 and 3.1.1. As we could not see any significant progress after one week, we decided to stop training these parameters and moved on and implement *Monte Carlo Tree Search* (MCTS) (see 3.4). We assume that the amount of parameters (3 x 56 indices, 3 PST per player) was too high and therefore much longer training time and a better GPU infrastructure would have been needed in order to effectively train the PST parameter. The value range of each index of the PST that we tried to train was also very large. Therefore we had to limit the random mutation offset to be in between the range of $[-200, 200]$. This limitation caused less variance and maybe prevented us from training more efficient PST, but was a necessary trade-off regarding the lack of having a GPU cluster infrastructure for the training. Furthermore, the probability of finding a good PST is very low as there are dependence between the list parameters.

3.4 Monte Carlo Tree Search (MCTS)

We had expected to have more success with our SPC, so we decided to try an algorithm which could improve performance without relying on a dataset. The MCTS is based on playouts. In each playout a MG game is played until the very end by selecting moves randomly. The final result of this game is used to weight the nodes so that better nodes are more likely to be chosen in future playouts [JKR17]. Due to time limitation, we only used MCTS for the first couple of moves as an *opening book* [NHI06] and then continued with the NegaScout algorithm.

3.5 Summary

Overall the results were not as good as we would have expected. As there was no training data we wanted to find out if SPC could lead to sufficient results. With a training period of only four weeks – including setup and implementation – we discovered that for MG only minimal improvement was possible. Furthermore, isolating and training parameters and then putting them back together lead to strange results sometimes. We believe this was caused by dependencies among the parameters. In total, none of the other groups was able to significantly increase their game AI performance using EL.

Yet, EL has the potential to be used to increase the performance of game AIs. Studies of Omid *et al.* show that by using EL for chess, good value functions can be derived [Da14]. In contrast to our work, they had an initial population of size 100, crossover rate was only 75% and mutation had a probability of 0.5%. They evaluated 200 generations. We believe they had better results with EL than we did, because the initial evaluation function of the population was derived from actuals moves of grandmaster-level games. As future work,

we would spend more time on MCTS. Assuming there were MG datasets, Koppel *et al.* [Da09] suggests to train a certain amount of AIs with the help of a fitness function. This fitness function was based upon the MG datasets. SPC was then performed on these AIs to determine the fittest one of them. This paper shows that SPC can be useful, if the initial population has been selected or trained before hand (and not random as we did).

4 Performance & Contest Results

In this chapter we will compare our performance during all three MS contests. We will mainly compare our results against the average performance of the other groups as a benchmark. We will not only measure our AI by counting the games that were won, but also have a look at the amount of rounds we were able to play and the minimum and maximum computation time per move. All together we will then try to measure the quality of our MG AI.

4.1 Overall Contest Performance

Overall our AI continuously improved during all three milestones. *Fig. 3 – left,* shows the amount of won games in percent (y-axis) relative to the total amount of played games for each MS (x-axis). Our AI did improve about 30% in between MS1 and MS2. On MS3 we had several different contest settings (Tournament mode, time per move, loading / no loading of opening books) and therefore a direct comparison is not possible.

Fig. 3 – right, shows the average amount of moves per game. In MS2 we were able to play five more moves per game on average compared to the other groups. As shown in chapter 4.1 the amount of won games was also improved in MS2. Therefore we conclude, that not only did we play longer games, but also win more games and therefore improved the performance of our AI.

4.2 Minimum & Maximum Time per Move

We also compared our minimum and maximum time per move with the average time of the other groups. *Fig. 4* shows the minimum (left) and maximum (right) computation time (y-axis) per MS (x-axis). We see, that our minimum time was 0 ms for every milestone contest. In contrast, our maximum computation time was below average which means we could have used our time more efficiently. In MS1 and MS2 we used approximately 5 ms per move, in MS3 we were able to triple the move selection time to 15,162 ms. Further improvements for our AI would therefore be the implementation of a more efficient usage of the computation time.

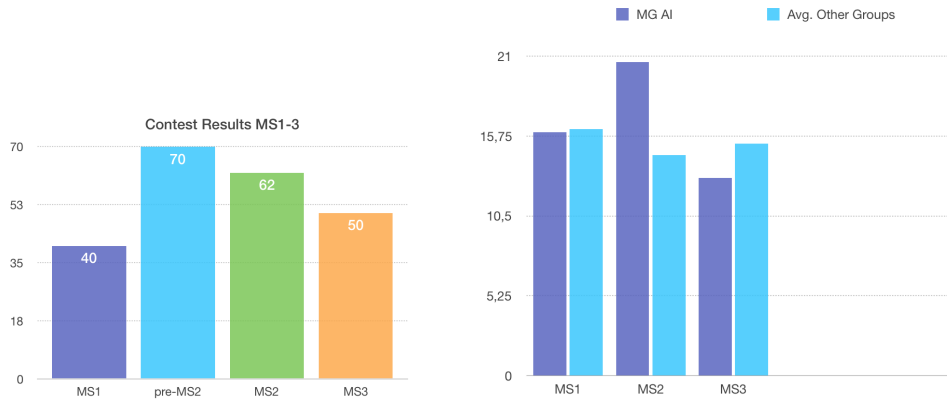

Fig. 3: Left: Average of won games in percent. Right: Amount of moves per game, MS1-3.

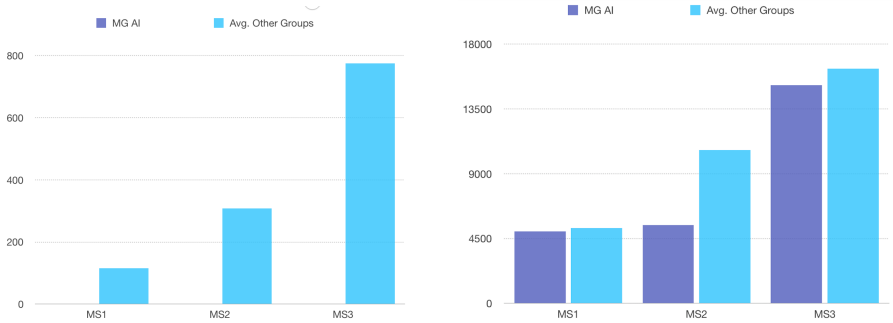

Fig. 4: Minimum (left) & maximum (right) time per move, MS1-3.

5 Conclusion

As shown in section 4.1 our AI was always performing on average or better than the other groups. The AIs we submitted at each MS were always competitive – they were able to win at least several games in each tournament. All the features that we implemented showed visible performance improvements. We reduced the amount of computed nodes using a combination of NegaScout and TT. Also, we could show in 4.2, how we adapted our time management to efficiently use the available time in each MS. For further work and improvements we would suggest speeding up the computation time in order to search deeper in the game tree. This could be realized by optimal selection of TT entries. For example, recognizing which positions are more likely to appear again. In addition, we would debug and complete our MCTS implementation in order to create more efficient opening book. Regarding the dynamic time management, we would try using a heuristic by using the branching factor to determine if we are able to search deeper.

Overall, the project showed us how important, but also difficult, it is to design an efficient value function. We also noticed that dependencies between different features can become very inefficient if there is a lack of computational power or time. The strongest game AIs, like AlphaZero, do not depend on manual value functions as they operate on zero game knowledge. It is important to find algorithms that work without data sets because we can not assume their availability in real life scenarios. Therefore, using AlphaZero on MG could be a promising alternative to the approach we followed in this project.

6 Acknowledgement

We want to thank our supervisor **Dr.-Ing. Stefan Fricke** for his great support and for giving us the highest possible flexibility during this project by letting us implement a wide range of different AI algorithms.

References

[Al90] AlthÃ¶fer, I.: An incremental negamax algorithm. Artificial Intelligence 43/1, pp. 57–65, 1990, ISSN: 0004-3702, URL: http://www.sciencedirect.com/science/article/pii/000437029090070G.

[BM01] Björnsson, Y.; Marsland, T. A.: Multi-cut alpha-beta-pruning in game-tree search. Theoretical Computer Science 252/, pp. 177–196, Feb. 2001.

[BM99] Björnsson, Y.; Marsland, T.: Multi-cut Pruning in Alpha-Beta Search. In (van den Herik, H. J.; Iida, H., eds.): Computers and Games. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 15–24, 1999, ISBN: 978-3-540-48957-3.

[BTW00] Baxter, J.; Tridgell, A.; Weaver, L.: Learning to Play Chess Using Temporal Differences. Machine Learning 40/3, pp. 243–263, Sept. 2000, ISSN: 1573-0565, URL: https://doi.org/10.1023/A:1007634325138.

[BUH70] Breuker, D.; Uiterwijk, J.; Herik, H.: Replacement Schemes for Transposition Tables. ICCA Journal 17/, Feb. 1970.

[CP20] (CPW), C. P. W.: Piece-Square Tables, https://www.chessprogramming.org/Piece-Square_Tables, [Online; accessed 11-May-2020], 2020.

[Da09] David, O. E.; van den Herik, H. J.; Koppel, M.; Netanyahu, N. S.: Simulating Human Grandmasters: Evolution and Coevolution of Evaluation Functions. 2009.

[Da14] David, O. E.; van den Herik, H. J.; Koppel, M.; Netanyahu, N. S.: Genetic Algorithms for Evolving Computer Chess Programs. IEEE Transactions on Evolutionary Computation 18/5, pp. 779–789, Oct. 2014, ISSN: 1089-778X.

[Da18] David, S.; Thomas, H.; Julian, S.; Demis, H.: AlphaZero: Shedding new light on chess, shogi, and Go, https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go, [Online; accessed 06-May-2020], 6.12.2018.

[Ge20] GeeksforGeeks: Tournament Selection (GA), https://www.geeksforgeeks.org/tournament-selection-ga/, [Online; accessed 11-May-2020], 2020.

[Go89] Goldberg, D. E.: Zen and the Art of Genetic Algorithms. In: Proceedings of the Third International Conference on Genetic Algorithms. Morgan Kaufmann Publishers Inc., George Mason University, USA, pp. 80–85, 1989, ISBN: 1558600063.

[JF16] Jinnai, Y.; Fukunaga, A.: Abstract Zobrist Hashing: An Efficient Work Distribution Method for Parallel Best-First Search. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. AAAI'16, AAAI Press, Phoenix, Arizona, pp. 717–723, 2016.

[JKR17] James, S.; Konidaris, G.; Rosman, B.: An Analysis of Monte Carlo Tree Search. In. Feb. 2017.

[Ma86] Marsland, T. A.: A Review of Game-Tree Pruning. J. Int. Comput. Games Assoc. 9/, pp. 3–19, 1986.

[NHI06] NAGASHIMA, J.; HASHIMOTO, T.; Iida, H.: SELF-PLAYING-BASED OPENING BOOK TUNING. New Mathematics and Natural Computation (NMNC) 02/, pp. 183–194, July 2006.

[Ph19] Phil, L.: Murus Gallicus, https://sites.google.com/site/theowlsnest02/home/murus-gallicus, [Online; accessed 11-May-2020], 2019.

[Po89] Polak, E.: Basics of Minimax Algorithms. In (Clarke, F. H.; Dem'yanov, V. F.; Giannessi, F., eds.): Nonsmooth Optimization and Related Topics. Springer US, Boston, MA, pp. 343–369, 1989, ISBN: 978-1-4757-6019-4, URL: https://doi.org/10.1007/978-1-4757-6019-4_20.

[RA10] Raoof, O.; Al-raweshidy, H.: Theory of Games: an Introduction. In. Sept. 2010, ISBN: 978-953-307-132-9.

[Ri01] Riechmann, T.: Genetic algorithm learning and evolutionary games. Journal of Economic Dynamics and Control 25/6, Computing, economic dynamics, and finance, pp. 1019–1037, 2001, ISSN: 0165-1889, URL: http://www.sciencedirect.com/science/article/pii/S016518890000066X.

[TD94] Thierens, D.; DE, G.: Elitist recombination: an integrated selection recombination GA. In. 508–512 vol.1, July 1994, ISBN: 0-7803-1899-4.