Comparison of Monte Carlo Tree Search Variants

Peter Guba[1,*], Jakub Gemrot[1]

[1]Charles University, Faculty of Mathematics and Physics, Ke Karlovu 3, 121 16 Praha 2, Czech Republic

Abstract

Monte Carlo Tree Search (MCTS) is a popular game AI algorithm that searches the state space of a game while using randomised simulations to evaluate new states. There have been many papers published about various adjustments of the original algorithm, however, work which compares these adjustments is rare. This lack of data can make it difficult to decide which algorithm to use without manual evaluation, which is time-consuming. The aim of this paper is therefore twofold. First, to create such a comparison, and second, to introduce a new variant, Weighted Propagation MCTS, which is based on the idea that one should be able to gather more information from a playout by taking into account all the states encountered during its computation. We chose three settings in which we compare our implementations - Chess, μRTS, and the 4X game Children of the Galaxy.

Keywords

Monte Carlo Tree Search, MCTS, games,

1. Introduction

The Monte Carlo Tree Search (MCTS) algorithm has pushed boundaries in a wide variety of problems [1, 2, 3, 4, 5, 6]. Since its introduction in 2006 [7], many adjustments have been proposed [8, 9], each claiming to outperform the original in some setting. However, we have been able to find only a few papers comparing multiple of these algorithms and only one of those in the context of a two-player sequential game. This presents a challenge for people seeking to make use of them, as the contexts in which these adjustments can be applied often have overlaps, and one cannot easily determine which variant best fits their purposes without reimplementing them for a common environment - a process that can be time- and resource-consuming.

In this paper, we address this gap by comparing eleven different MCTS adjustments. Ten of these are taken from papers by other authors, while the eleventh, called Weighted Propagation MCTS, is our own variant, which adjusts the way that playouts are evaluated.

For our testing environments, we chose three games - chess, the real-time-strategy (RTS) computer game simulator μRTS [10], and a 4X turn-based computer game called Children of the Galaxy [11].

The rest of this paper is structured as follows: In Section 2, we present some related work. In Section 3, we present a short overview of how the original MCTS algorithm works. In Section 4, we briefly describe the workings of the MCTS variants that we compare. In Section 5, we present the workings of our testing environments. In Section 6, we introduce script-based search - a restriction of traditional action-space search used to shrink the branching factor of our search. In Section 7, we report our experiments and their results. In the final Section, we offer some conclusions and describe potential subjects for future work.

2. Related Work

Since MCTS does not rely on any game-specific knowledge, it has been successfully used in a wide variety of games, such as go [4], Total War: Rome II, Ms. Pac-man [3], and Poker [1].

ITAT'25: Information Technologies – Applications and Theory, September 26–30, 2025, Telgárt, Slovakia

*Corresponding author.

guba@ksvi.mff.cuni.cz (P. Guba); gemrot@gamedev.cuni.cz (J. Gemrot)

0000-0002-3640-5152 (P. Guba); 0000-0002-4394-3450 (J. Gemrot)

Due to its practical usefulness and theoretical properties, the algorithm has been a long-standing subject of academic interest, with two surveys being published that list numerous adjustments of the original algorithm that have been proposed since [8, 9].

Our literature review found four papers that compare a higher number of MCTS variants [12, 13, 14, 15]. The first two do so in the context of single-player games from the general video game AI (GVG-AI) framework, and the third was done on simultaneous-move games. There are two main differences between these three papers and ours. First, we conduct our experiments on sequential two-player games and second, we allow for the use of domain-specific knowledge.

The fourth paper, by Hsueh et al., compares 4 different variants and their combinations in Chinese dark chess. This is a sequential two-player game, similar to the ones we use, and the authors also allowed for the use of heuristics. Their work is therefore similar to ours, but whereas their aim was to improve on an agent for playing Chinese dark chess, ours is to create a more general comparison, which is why we use multiple games.

3. MCTS

MCTS explores the state-action space of a game while trying to balance the exploration of new moves and the exploitation of moves already deemed good. It does this by building a partial search tree where each node corresponds to a game state. It operates in four steps, which repeat in a loop until a terminal condition is reached and then picks a move. The individual steps work as follows.

- **Selection** The algorithm traverses the already explored part of the search tree from the root until it reaches a node with children that have not been visited yet. At every node, it decides which action to perform using a strategy called the *tree policy*.
- **Expansion** A new child node is created and appended to the node selected in the previous step.
- **Simulation** A simulation called a playout is run from the newly created node. The moves in the simulation are made according to the algorithm's *default policy* until a terminal state is reached or a cut-off condition is triggered. A numeric score is obtained from the last reached state.
- **Backpropagation** The score is then used to adjust the scores of nodes within the partial search tree on the path from the newly created node to the root node.
- **Termination** The four steps described above repeat until some terminal condition is reached - in most applications, this is an exceeding of the time limit. Afterwards, a move is picked using some strategy that is usually different from the tree policy, as exploration no longer plays a role.

4. MCTS Variants

Our original motivation was to improve combat AI in the game Children of the Galaxy (described in Section 5.2). To that end, we searched for MCTS variants to implement in a number of papers, including an extensive survey of MCTS techniques [8]. The only selection criterion we used (besides having to be adjustments of MCTS) was whether they were applicable to our environment.

Due to space constraints, we only present short overviews of the chosen variants here; for details, we refer the reader to the original papers.

4.1. Upper Confidence bounds applied to Trees[16]

Upper Confidence bounds applied to Trees (UCT) is the most well-known variant of MCTS. It balances exploration and exploitation using the UCB1 policy, which picks the node that maximises

$$\bar{X}_i + c\sqrt{\frac{\log s_p}{s_i}} \tag{1}$$

where $\bar{X}_i$ is the current mean reward at node i, s_i is the number of visits of that node, s_p is the number of visits of its parent and c is a tunable parameter.

This formula was developed for the Multi-Armed Bandit (MAB) problem where it is popular due to its regret bounds. As the decision made at every node during the selection step can be considered an instance of MAB, it can be applied here.

UCT's default policy consists of picking random moves. Its evaluation function assigns the final states of playouts a score of 1, 0, or 0.5, depending on whether the result was a win, loss, or draw/the game did not finish. The algorithm returns the move with the best average score. The parameter c for the UCB formula was set to 1.

Note that the default policy, way of picking moves, and way of evaluating final states used here are not a part of UCT's specification – UCT only defines the tree policy. We decided to go with these approaches because they can be considered the most standard implementations of these parts of MCTS, just like UCB1 can be considered the most standard implementation of its tree policy.

All the following algorithms are based on this version of UCT and only differ in explicitly stated ways.

4.2. SR+CR MCTS [17]

SR and CR in the name of this variant stand for "simple regret" and "cumulative regret" respectively. These are measures of the difference between the reward for an optimal decision and the reward for a decision made based on some strategy . Simple regret is defined as follows:

$$r_n = \mu^* - \mu_{x_n} \tag{2}$$

where μ^* is the expected reward for choosing the optimal node, x_n is the node that would be picked at iteration n and μ_{x_n} is its expected reward. It is the regret for choosing a suboptimal action at round n. Cumulative regret, on the other hand, is computed over all rounds.

$$R_n = \sum_{t=1}^{n} \mu^* - \mu_{x_t} \tag{3}$$

The UCB1 formula that is used by UCT aims to minimise the cumulative regret of all the arm pulls in an MAB setting. This causes the algorithm to exploit the best moves at the root a lot more than it needs to, since it only actually picks a move to play at the end of its run. To remedy this issue, this variant uses a policy at the root that minimises simple regret (otherwise, it uses UCB1).

An example of such a policy is the ϵ-greedy policy, which picks the current best move with probability ϵ and any other with probability $\frac{1-\epsilon}{N-1}$, where N is the number of moves. Unlike UCB1, it doesn't require every child of a node to be tried once before it can be applied, allowing early exploitation.

4.3. VOI-aware MCTS [17]

Like SR+CR MCTS, this variant tries to minimise simple regret when picking actions at the root. It does this by approximating the value of information (VOI) that can be gained from choosing different actions. As this is impossible to compute exactly, the authors used the myopic assumption that the algorithm will only sample one of the available actions and used it to create the following equations for approximating the VOI of different nodes:

$$VOI_\alpha = \frac{\bar{X}_\beta}{n_\alpha + 1} \exp(-2(\bar{X}_\alpha - \bar{X}_\beta)^2 n_\alpha) \tag{4}$$

$$VOI_i = \frac{1 - \bar{X}_\alpha}{n_i + 1} \exp(-2(\bar{X}_\alpha - \bar{X}_i)^2 n_i), i \neq \alpha \tag{5}$$

where $\alpha = \text{argmax}_i \bar{X}_i$, $\beta = \text{argmax}_{i,i \neq \alpha} \bar{X}_i$, that is, they denote the current best and second best actions, respectively. The node with the highest VOI is always selected at the root and UCB1 is used otherwise.

For the details of the derivation of these equations, please see the original paper, as it is beyond the scope of this work.

4.4. UCB1-Tuned MCTS [18]

UCB1-Tuned is a formula that provides tighter bounds on the uncertainty of observations than UCB1 using an estimate of the rewards' variance. This variant uses it as its tree policy.

4.5. Sigmoid MCTS [19]

This variant combines two common ways of evaluating playouts - win-or-lose and final score. The former is the one used in our UCT implementation. In the latter, some more complex metric is used, which gives a more nuanced evaluation of state quality. To combine them, the authors apply a sigmoid function to the final score.

$$f(x) = \frac{1}{1 + e^{-kx}} \tag{6}$$

This function changes the value to something between win-or-lose and final score. How close it is to either is determined by the constant parameter k.

4.6. Relative Bonus MCTS [20]

Instead of just backpropagating an evaluation of a playout's final state, this variant applies a bonus to it computed from its length and the depth at which it starts. The idea is that the longer the playout, the less reliable information is obtained. A sigmoid function, the shape of which can again be altered using a parameter labelled k, is used when computing the bonus to bound and shape its values.

4.7. Qualitative Bonus MCTS [20]

Like Relative Bonus MCTS, but the bonus is derived from the quality of the last state of the playout.

4.8. Relative-Qualitative Bonus MCTS [20]

A combination of the previous two variants - both bonuses are computed and added to the playout score. The sigmoid functions for both bonuses uses the same value of k.

4.9. MCTS-HP [11]

Like Sigmoid MCTS and Qualitative Bonus MCTS, this variant utilizes some more nuanced metric for state quality evaluation. The score obtained from the evaluation is then backpropagated and normalised at every node by the maximum possible value of the metric in that node.

4.10. Feedback Adjustment Policy MCTS [21]

This variant partitions playouts into groups based on the time at which they were computed (the later the better) and assigns a multiplicative factor n to each group (every playout from a group is considered to be worth n playouts). The partitioning is based on the assumption that playouts that are performed later offer more information.

The authors suggested two schemes for both the partitioning of playouts and the computation of multiplicative factors - linear (lin) and exponential (exp). The schemes for the two tasks are independent

(i.e. one can use a linear scheme for partitioning and an exponential scheme to compute the multiplicative factors, or vice versa).

It is important to note that, unlike the other variants, this one relies on knowing the number of playouts beforehand - information that is not available when the algorithm's execution is bounded by a time limit.

4.11. Weighted Propagation MCTS

Our proposed variant, Weighted Propagation (WP) MCTS, is based on the idea that if a good heuristic for evaluating states is available, it should be possible to extract more useful information out of a playout than just the value of its final state by taking into account how the value of states evolved through time.

To put this formally, a playout is a sequence of states $s_1, s_2...s_n$. In standard MCTS, the evaluation function is of the form $f(s_n)$. We propose to use a function that takes into account all the encountered states - $f(s_1, s_2...s_n)$. In this paper, we specifically use

$$f(s_1, s_2...s_n) = \frac{\Sigma_{i=1}^{n} w(i,n) \cdot h(s_i)}{u \cdot \Sigma_{i=1}^{n} w(i,n)} \tag{7}$$

where $h(s_i)$ is a heuristic function used to evaluate states, $w(i,n)$ is a weight function computed from the position of the state in the playout, and u is an upper bound on the heuristic value of a state. The upper bound is computed every time the algorithm is started as the maximum heuristic value of the state if one player's pieces/units are discarded. This corresponds to the player destroying their opponent without suffering any losses and ensures that the values are always between 0 and 1.

Our weight function takes into account two criteria - first, the later a state occurs in a playout, the less probable it is to actually be encountered. Second, the earlier a state occurs in a playout, the less information it gives us about how the game can evolve. Our function can therefore be separated into two functions, which are then averaged.

$$w(i,n) = \frac{w_1(i) + w_2(i,n)}{2} \tag{8}$$

Both of these functions are exponentials with the base determined by a parameter of the algorithm and the exponent determined by the data.

$$w_1(x) = \min\left(b_1^{-x}, 10000\right) \tag{9}$$

$$w_2(x,y) = \min\left(b_2^{x-y}, 10000\right) \tag{10}$$

The 10000 upper bound on the weights was added as small bases could otherwise cause overflow.

5. Test Environments

We implemented and tested all presented MCTS variants in 3 environments with different characteristics: chess (turn-based, small branching factor), Children of the Galaxy (turn-based, large branching factor), and μRTS (real-time, large branching factor).

The code for these environments, as well as for some auxiliary programs we used, can be found in our GitHub repository [1].

[1] https://github.com/peter-guba/MCTSTesting

Algorithm Name	Abbreviation	Chess	CotG	µRTS
FAP MCTS	FAP	segmentation: exp multiplication: lin N: 100	segmentation: exp multiplication: lin N: 100	segmentation: lin multiplication: lin N: 100
MCTS-HP	HP	X	X	X
Qualitative Bonus MCTS	QB	k = 100	k = 0.1	k = 0.1
Relative Bonus MCTS	RB	k = 10	k = 0.5	k = 100
Relative-Qualitative Bonus MCTS	RQB	k = 1	k = 0.1	k = 10
Sigmoid MCTS	Sig	k = 1	k = 0.1	k = 0.1
SR+CR MCTS	SR	policy: ϵ-greedy ϵ = 0.75	policy: ϵ-greedy ϵ = 0.75	policy: ϵ-greedy ϵ = 0.75
UCT	UCT	X	X	X
UCB1-Tuned MCTS	U-T	X	X	X
VOI-aware MCTS	VOI	X	X	X
WP MCTS	WP	b_1 = 10, b_2 = 10	b_1 = 2, b_2 = 2	b_1 = 2, b_2 = 1

Figure 1: The parameter values of all the tested algorithms in every testing environment, as well as their abbreviations.

5.1. Chess

We assume that chess is a well-enough-known game to need no introduction. We will therefore only explain the way we implemented the final score heuristic mentioned in Section refsigmoid and also used in Sections 4.7, 4.8, 4.9 and 4.11. All pieces are assigned static values, which are summed up for every player and the difference between these two sums is computed. We assigned the following values: 100 for pawn, 300 for knight, 320 for bishop, 500 for rook, 900 for queen and 1200 for king.

5.2. Children of the Galaxy (CotG)

CotG belongs to a subgenre of strategy games called 4X games. In the game, the player tries to conquer a galaxy using various means, from technological research to military expansion. It can be divided into discrete subtasks, of which we target only one - combat between units. This is the most critical part of CotG, as any higher-level decision making (where to attack, where to defend, etc.) is bounded by the quality of combat outcome estimations.

Like chess, CotG is divided into discrete turns. Its matches are carried out on a hexagonal grid and involve units that the player can manipulate. Each unit has a certain number of hit points (HP) as well as an attack range. Once an enemy unit is within a unit's attack range, the unit can attack it, thereby decreasing its HP. When this reaches 0, it is removed from the game. The game ends when at least one of the players has no more units. In each turn, the player can move as many units as they like, but each unit only has a limited distance by which it can move in one turn.

For our experiments, we used a CotG combat simulator created by Šmejkal [11]. The final score heuristic implementation here is similar to the one we used in chess, except instead of piece values we use the remaining HP of units.

5.3. µRTS [22]

µRTS is a simple real-time strategy (RTS) game implementation created by Ontañón and intended for reinforcement learning experiments that involve RTS games. The game is played in real time and, similar to CotG, requires balancing between various tasks, such as resource gathering, fighting, and research. Again, each of these tasks can be controlled by a separate system, therefore we only apply the tested algorithms to combat, which in this game takes place on a square grid. Like in CotG, it involves the player moving units that can attack one another, and the game ends when at least one of the players has no more units.

Due to its real-time nature, we had to alter our algorithms to take action durations into account. We did this using the approach described by Churchill and Buro in [23].

The final score heuristic used here is the same as the one used in CotG.

6. Script-based search

Strategy games where players battle using large numbers of units are known to have large branching factors, making them very challenging for algorithms like MCTS, which rely on traversing a substantial portion of the game tree. One technique for addressing this is restricting possible unit actions using scripts that process the current game state and output a small number of actions (usually only one) for a given unit to execute next [23, 24]. We chose this approach instead of using an adaptation of MCTS, such as NaiveMCTS [10], as we wanted to keep the algorithms consistent across all environments. For our scripts, we chose No-Overkill-Attack-Value (NOKAV) and Kiter, as these are commonly used for creating these kinds of action abstractions.

In CotG, the scripts are chosen at every turn and then discarded. In μRTS, they are followed until they are either completed or determined impossible to follow further (for example if a unit's target is destroyed).

7. Experiments

Our aim was to determine whether some of the discussed MCTS variants could be said to perform better than others. To do this, we created tests in which we pitted them against each other and, for each game, ran them in a round-robin tournament while gathering data about their performance. In the following subsections, we describe how our tests and whole experiment were designed, the specific settings, values, and tools we used, and finally, we present our results.

7.1. Test Design

Our tests consisted of a series of 1 vs 1 matches. Each test was parameterised by two algorithms with specified parameter settings and a playout setting that was identical for both algorithms. We selected 3 playout settings - 1000, 5000, and 10000.

The same tests were run in each environment with a limit on the number of rounds (for chess and CotG) or game ticks (for μRTS) set to 1000. We played 18 matches per test.

In CotG and μRTS, these matches were separated into 3 categories based on the number of units that each team had at its disposal - 4 vs 4, 8 vs 8, and 16 vs 16 - and multiple matches were conducted in each category to get more reliable results. We refer to the number of units as the *combat setting*. In chess, we ran the same number of matches as in the other two environments, but all were started from the default chess starting position.

The reason why we decided to restrict playouts rather than time was that such results are independent of the underlying software (the operating system and its configuration) and hardware they are run on. Any paper published now about MCTS that restricts algorithm runtime will be outdated in ten years, while results based on playout restrictions will remain replicable. This approach should also ensure that the agents perform the same on all computers.

It could be argued that this unfairly advantages our algorithm, as it should be one of the most computationally demanding ones. We therefore conducted an extra set of experiments, identical to the ones described above, except with varying numbers of playouts for WP MCTS.

7.2. Experiment Design

Due to time constraints and the problem of combinatorial expansion, we had to limit the number of experiments. We started by listing all the possible algorithm settings we wanted to try (by algorithm setting, we mean an algorithm with all its parameters specified), producing 117. We then tested each of

these against UCT, which, as mentioned in Section 4.1, can be considered the default implementation of MCTS. We ran these tests in each of our testing environments. Based on the results, we picked one algorithm setting for every MCTS variant and every test environment (so the algorithm settings used in different environments were allowed to differ) and tested every possible pair.

In the second round of testing, we ran 56 repetitions of every test in every environment. This gave us 3 environments * 3 playout settings * 55 algorithm pairings * 56 repetitions = 27,720 tests (498,960 matches) in total.

7.3. Experiment Specification

The chosen parameter values for each algorithm are shown in Figure 1, as well as the abbreviations that we will be using. X indicates there were no parameters to set.

Our experiments were run using MetaCentrum [25] - a Czech virtual organisation that offers resources for grid computing. We created test files, each of which contained 3 tests with the same algorithms but different playout settings, and assigned each of the tests one processor and up to 85 GB of memory, depending on the testing environment. The chess and CotG testing environments were written in C# targeting .NET 6. μRTS was written in Java using JDK 8.

We now go over experiment details specific to each environment.

7.3.1. Chess

All games were started from the default chess starting position. The player who moved first alternated between matches. A match was considered finished when one of the standard chess terminating conditions occurred or when the round limit was reached. If the match terminated without either of the kings being captured, it was labelled a draw.

7.3.2. CotG

We used two types of units - *Destroyers* and *Battleships*. The former is a unit with high damage but short attack range and low HP, while the latter has medium attack range, lower damage, but high HP and shields. While the game offers other units, these two provided enough diversity for our purposes.

In every combat setting, half of the units were always of one type and half of the other. Their positions were randomly generated for one team and for each combat setting beforehand, symmetrically assigned for the other team and then remained constant.

The first player alternated between matches. A match was considered over either when all the units of at least one of the players were destroyed, or the round limit was exceeded. Unlike in chess, exceeding this limit here did not necessarily mean a draw. Instead, the winner was decided based on the remaining HP of the players - the player with more HP was labelled the winner.

7.3.3. μRTS

The setup in this environment was quite similar to CotG. We also used two types of units - *heavy* (slower, higher damage) and *light* (faster, lower damage). Half the units were again assigned each type, and their positions were randomly generated for one side, symmetrically assigned for the other and then remained constant.

As both players move at the same time in μRTS, there was no switching of who got to go first.

The terminal conditions and way of determining the winner were the same as in CotG.

7.4. Results

For all three testing environments, we show the numbers of wins of every algorithm against every other algorithm. For CotG and μRTS, we also show the accumulated numbers of wins for every algorithm and combat setting. We omit splitting the results by number of playouts, as that did not show any significant differences in algorithm rankings.

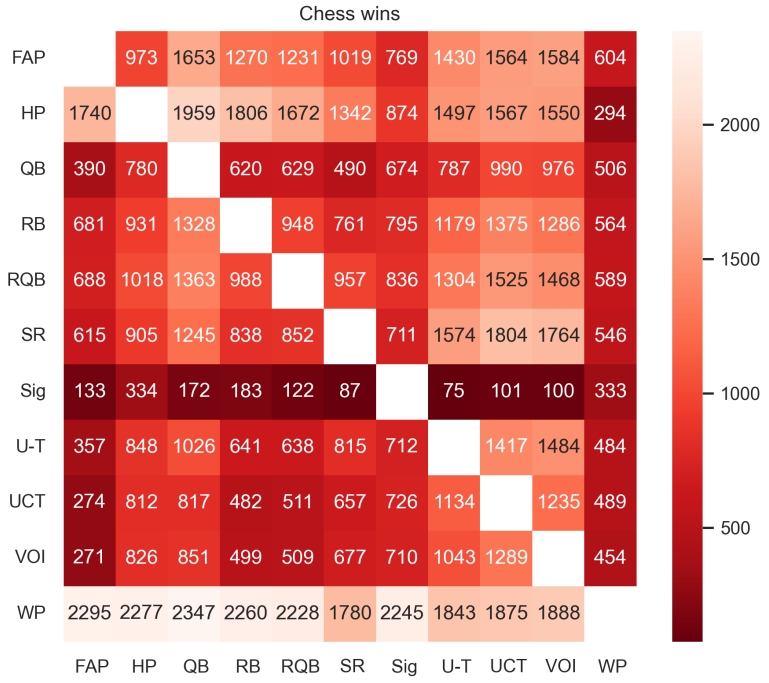


Figure 2: The number of wins of every MCTS variant against every other in chess. There were 3024 games played per pairing.

7.4.1. Chess

The number of wins scored by algorithms against one another in chess can be seen in Figure 2. As there were no draws, we omit showing them. There are two notable performers. The first is Sigmoid MCTS, which always performed poorly. This is unsurprising, as, unlike in CotG and μRTS, the winning condition here is not eliminating all the opponent's pieces. The heuristic that we used here therefore misleads the algorithm.

The other is WP MCTS, which achieved a high win rate against every other variant. Other well-performing variants are MCTS-HP and FAP MCTS. It is interesting to note that WP MCTS actually scored better against these than against some overall worse-performing ones, including UCT. It may be that the moves these variants choose are more similar to those WP MCTS does, and it is therefore able to better model them.

Also noteworthy is the fact that, although every algorithm except Sigmoid MCTS beat UCT in one-on-one matches, some variants, like Qualitative Bonus MCTS and VOI-aware MCTS, scored fewer wins overall.

7.4.2. CotG

The top part of Figure 3 shows the number of wins scored by every algorithm against every other algorithm in CotG. The data here is more evenly distributed than in chess, which is expected as there is often less of a clear distinction between optimal and suboptimal moves in CotG. However, like in chess, there were no draws.

The worst-performing variants are UCT, UCB1-Tuned MCTS, and VOI-aware MCTS. In the case of UCT, this is to be expected, as every other variant was designed to be an improvement over it. The best-performing variants, on the other hand, are WP MCTS, MCTS-HP, and Sigmoid MCTS. These rely on the final score metric described in Section 5.2, which evidently provides much useful information in this context.

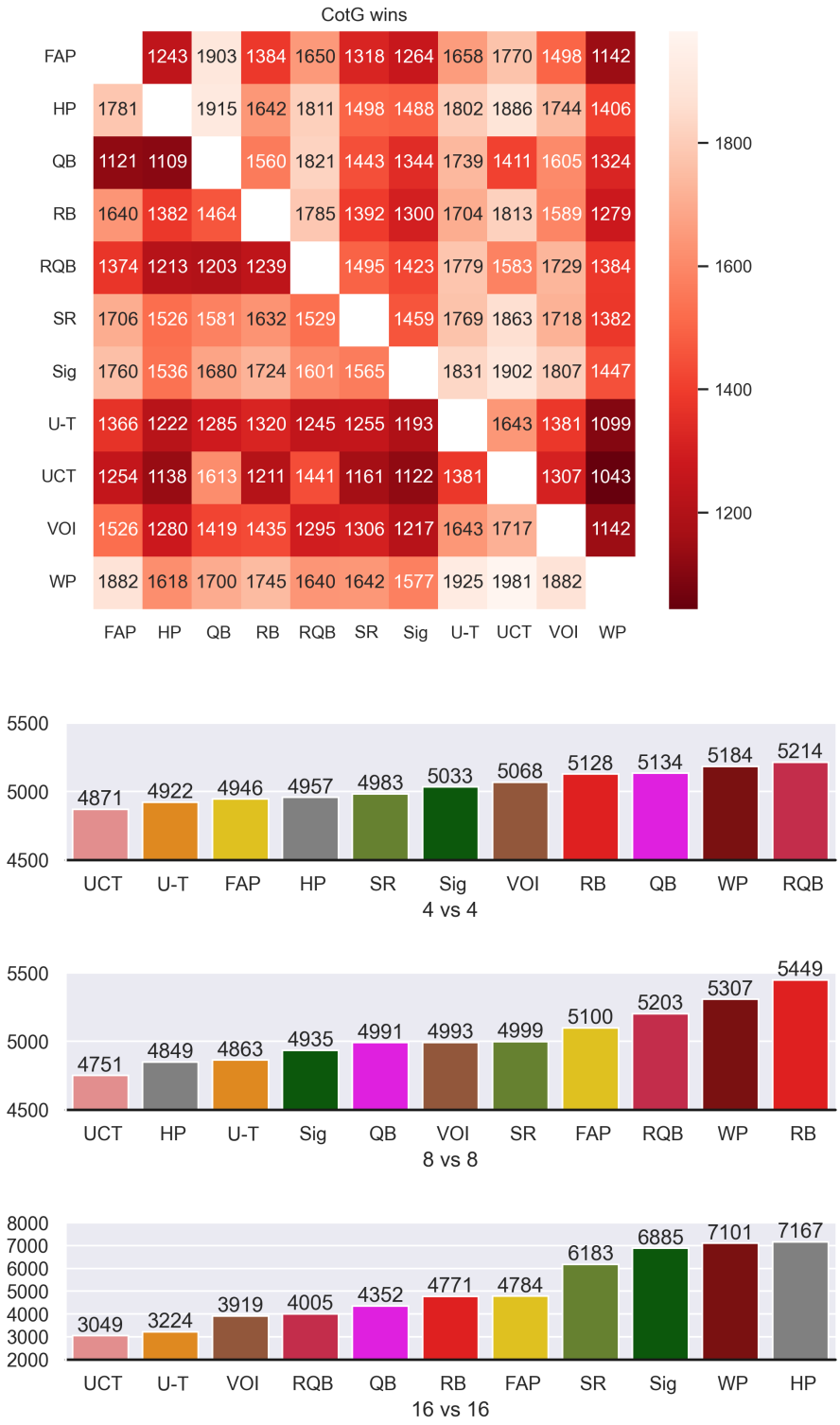


Figure 3: (top) The number of wins of every MCTS variant against every other in CotG. There were 3024 games played per pairing. **(bottom)** The accumulated number of wins of every algorithm in every combat setting in CotG. There were 10080 games run per algorithm and combat setting.

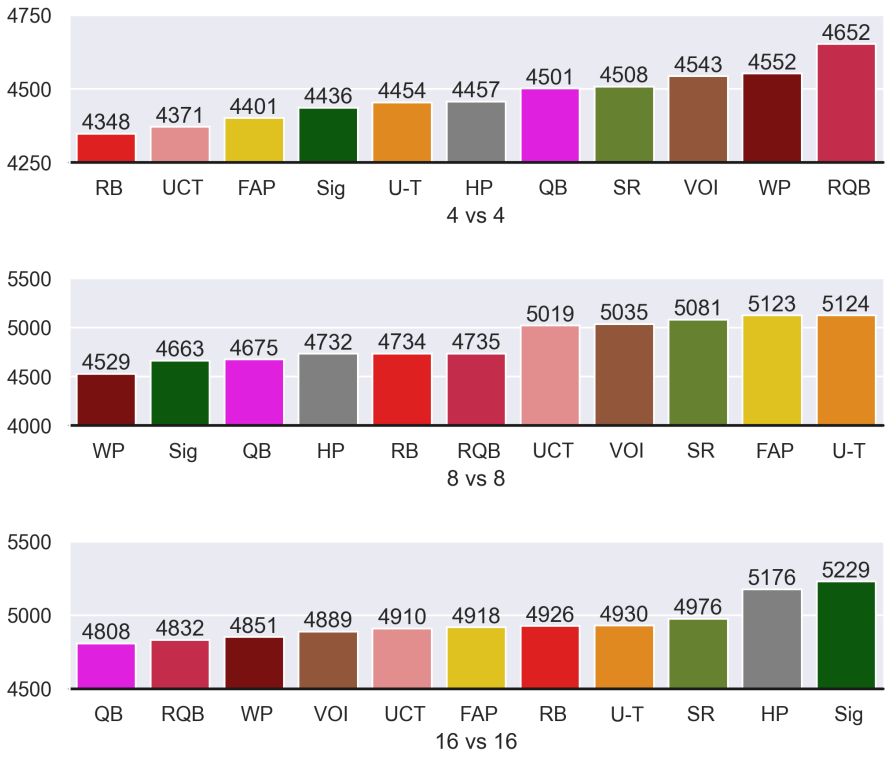


Figure 4: The accumulated number of wins of every algorithm in every combat setting in µRTS. There were 10080 games run per algorithm and combat setting.

The bottom part of Figure 3 shows the number of wins of each algorithm in each combat setting. In the first two, the results are quite evenly distributed. This indicates that if the algorithms can sufficiently explore the search tree in a game with many equally good moves, their differences diminish. In the last combat setting, the differences suddenly get much more pronounced. We think this is because the search trees are so large that, in the beginning, the algorithms cannot even properly explore their first levels, therefore the ones that can extract the most information out of a single playout and the ones that do not need to finish searching the first level before proceeding to the next fare much better.

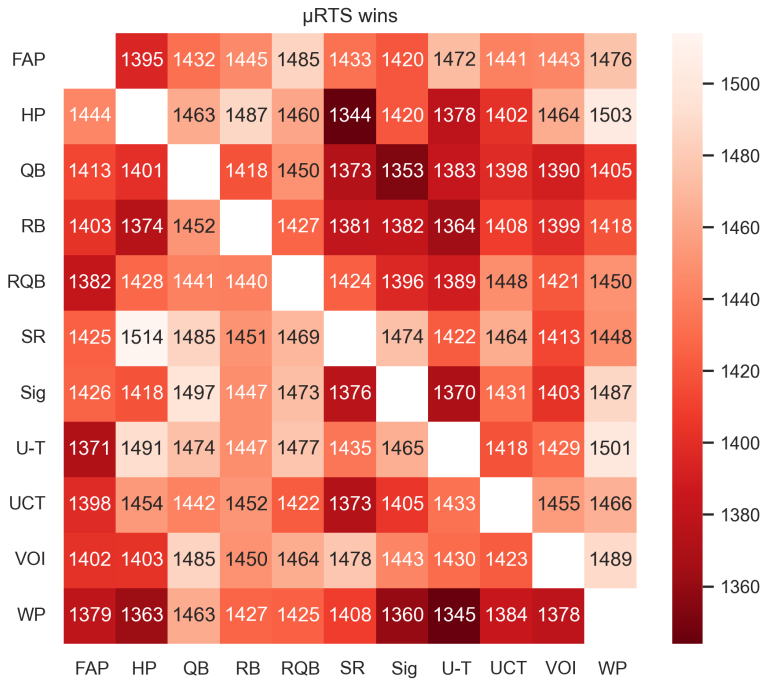
7.4.3. µRTS

In µRTS, the results were even more evenly spaced than in CotG. As seen in the top part of Figure 5, the difference between the worst result and the best one is only 170 points, far smaller than in CotG (879) or chess (2302). What's more, unlike in chess and CotG, some number of matches ended in draws in this setting, as shown in the bottom part of Figure 5. This could be due to the persistency of script assignments, as mentioned in Section 6 - since scripts in µRTS are followed until they are completed or deemed impossible to follow further, it is possible that they are picked much less frequently, which means smaller search trees which can be searched faster and allow for less strategising.

The data in Figure 4 is quite evenly spaced in all combat settings, unlike in CotG. The ordering of the algorithms also differs greatly from one combat setting to the next, leading us to believe that the tested MCTS variants are quite equal, but perform slightly better or worse depending on conditions such as the initial unit positions.

Our algorithm performed poorly in this environment, scoring the lowest number of wins overall. The reasons for this are unclear, especially in the 16 vs 16 combat setting, where MCTS-HP and Sigmoid MCTS - two other variants that use the same heuristic for evaluating states as WP MCTS - performed the best.

One possibility is that the heuristic we use is biased, and in smaller search trees which the other



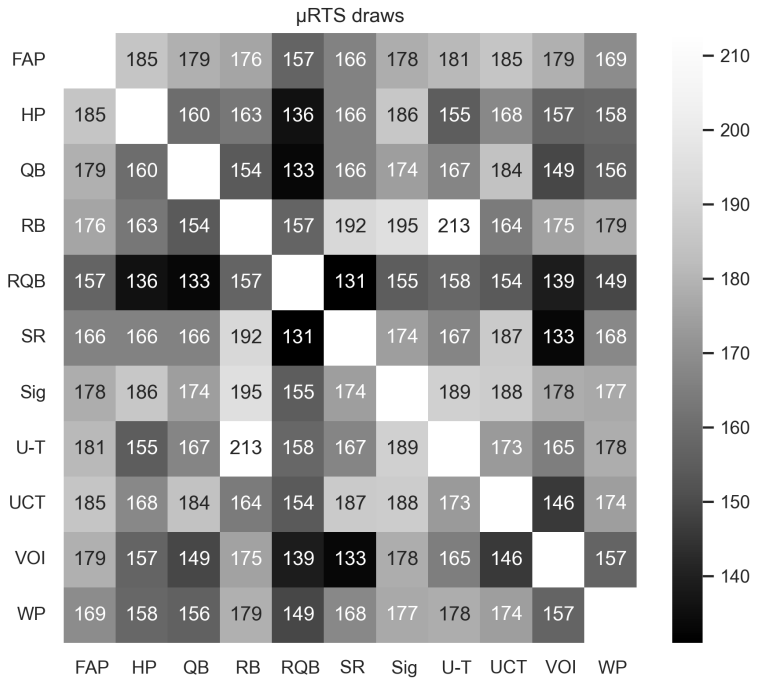


Figure 5: (top) The number of wins of every MCTS variant against every other in µRTS. **(bottom)** The number of draws of every MCTS variant against every other in µRTS. There were 3024 games played per pairing.

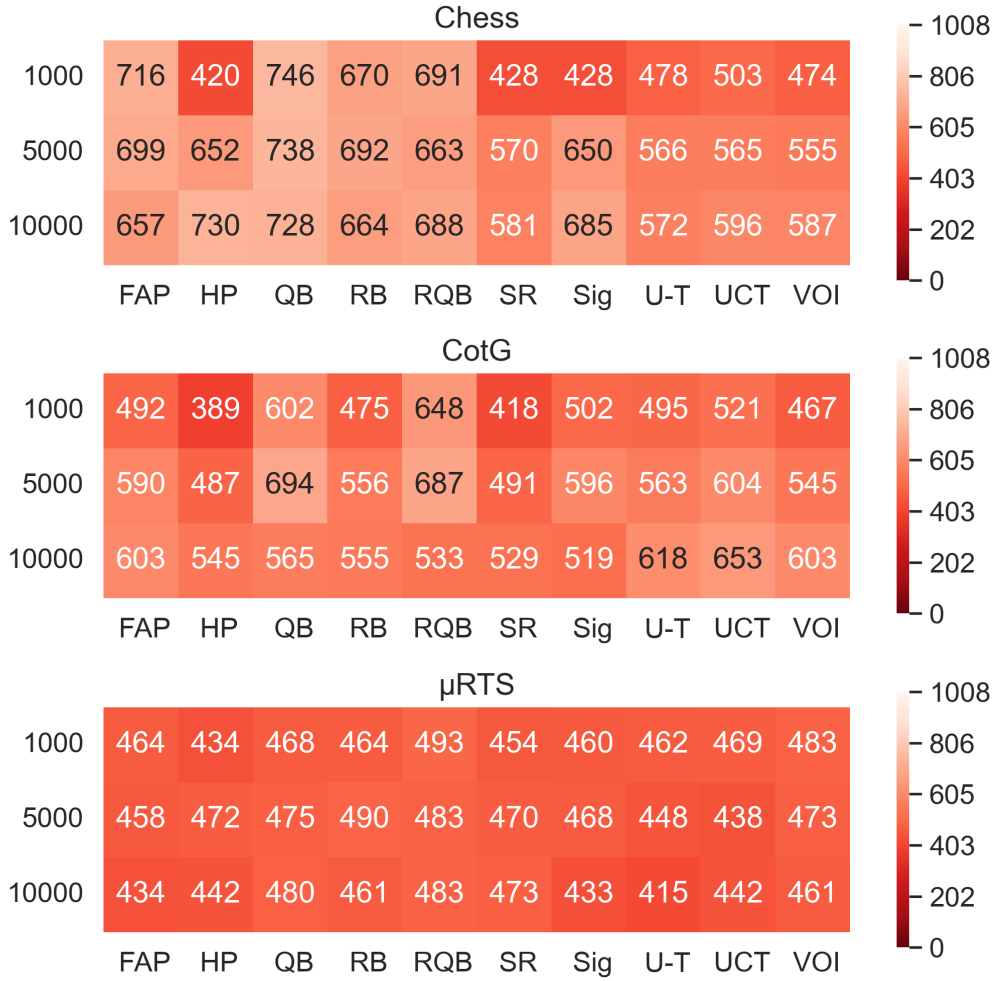


Figure 6: The number of wins of WP MCTS against every other variant in all testing environments at different playout settings (different just for WP MCTS). There were 1008 games played per pair of algorithms and playout setting.

algorithms can sufficiently explore, they can find better moves. If that were the case, however, we would expect to see the same behaviour in CotG, which was not observed. More research is required to identify the cause with certainty.

7.4.4. Extra WP MCTS tests

Figures 6 and 7 show the results of WP MCTS in the extra tests that we ran to see how well it would fair with a reduced number of playouts (against the other algorithms with 10,000).

In chess, cutting the number of playouts down to 5000 did not show a significant decrease in win rate. At 1000 playouts, the algorithm's performance clearly declined, but it nonetheless won more times than it lost in every pairing.

In CotG, giving the algorithm 5000 playouts actually produced an improvement in some cases, though this could be a random fluctuation. Only 2 of the other variants managed to outperform it at this setting. At 1000 playouts, the algorithm's win rate decreased for most pairings, though it still outperformed 3 out of the 10 tested variants.

In μRTS, performance was quite uniform independent of the number of playouts, and the algorithm actually fared better at 1000 playouts than at 10,000. This further supports our hypothesis that the search trees here are smaller and offer fewer opportunities for strategising.

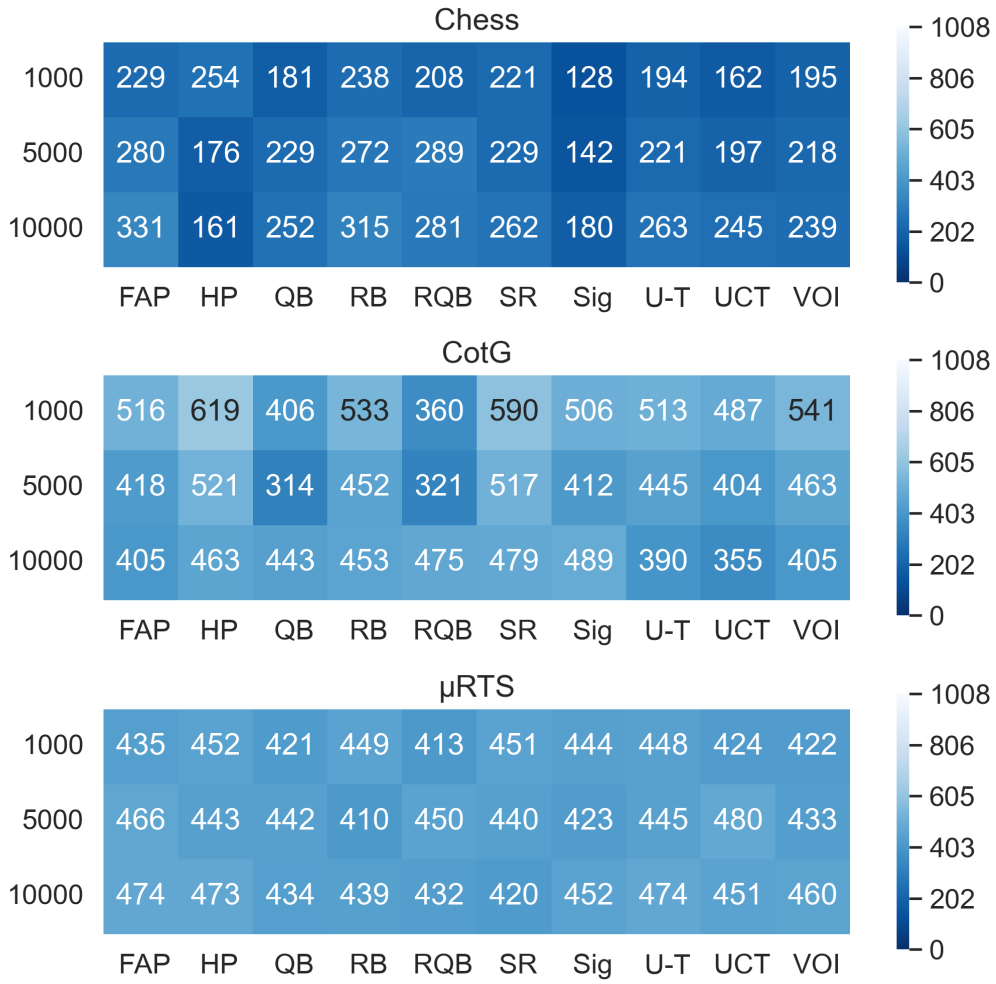


Figure 7: The number of losses of WP MCTS against every other variant in all testing environments at different playout settings (different just for WP MCTS). There were 1008 games played per pair of algorithms and playout setting.

7.4.5. Overall assessments

- FAP MCTS - One of the most robust variants, though never one of the best-performing ones. Does not seem to perform well in combat settings with a small number of units, though this could be due to us using the same parameter values in all combat settings.
- MCTS-HP - Also a very robust variant, performed very well in every testing environment.
- Qualitative Bonus MCTS - One of the worst-performing variants, did not achieve any noteworthy results. This is surprising because it uses the same metrics for evaluating game states as MCTS-HP, WP MCTS and Sigmoid MCTS, which all outperformed it significantly.

- Relative Bonus MCTS - Average performing variant, also did not achieve any noteworthy results.
- Relative-Qualitative Bonus MCTS - Also average performing. Strangely, it often performed slightly worse than just Relative Bonus MCTS. Perhaps having separate sigmoid functions for the two bonuses could provide an improvement.
- Sigmoid MCTS - Due to the heuristic used to evaluate states, it did not perform well in chess, but otherwise, it achieved a high win rate. Still, if a heuristic for judging state quality is available, MCTS-HP or WP MCTS seem like better choices.
- SR+CR MCTS - The most robust variant. Going by the total number of wins, it was among the top 4 in every environment. It was also the only variant for which we used the same setting in every environment after the initial round of testing.

- UCB1-Tuned MCTS - Usually achieved very similar results to UCT. As it only has a slight adjustment of the UCB formula, that is unsurprising.
- VOI-aware MCTS - Performed surprisingly poorly, given that the motivation behind it is the same as for SR+CR MCTS and in the original paper, it even outperformed this variant. Its disadvantage may lie in the fact that, like the other variants, it needs to fully expand the root node before moving on to the next level of the tree.
- WP MCTS - Performed the best in chess and CotG, the worst in μRTS. However, as the range of results achieved in μRTS was quite small, we do not consider this a substantial failure.

8. Conclusions and Future Work

In this paper, we have compared 11 different variants of the Monte Carlo Tree Search algorithm in three different environments. One of these was a novel algorithm called WP MCTS, and it proved to be the best-performing algorithm in two of the environments.

As the environments used in this paper are freely available on our GitHub, they can be used by other researchers, and their results will be comparable to ours.

For future work, we might further explore aspects of these algorithms, such as their behaviour in situations where they control even more units, their performance in other environments, or how the optimal algorithm settings change with different number of playouts. We would also like to investigate further the lack of variability of our results in μRTS.

Regarding our algorithm, we would like to look into how well it performs equipped with different weight and evaluation functions, as well as further explore the idea of getting the most out of playouts by analysing all of their states.

Acknowledgments

Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

References

[1] G. V. D. Broeck, K. Driessens, J. Ramon, Monte-Carlo Tree Search in Poker using expected reward distributions, in: Asian Conference on Machine Learning, 2009, pp. 367–381.

[2] T. Kozelek, Methods of MCTS and the game Arimaa, Master's thesis, Charles University, 2009.

[3] T. Pepels, M. H. Winands, M. Lanctot, Real-time Monte Carlo Tree Search in Ms Pac-Man, Transactions on Computational Intelligence and AI in games 6 (2014) 245–257.

[4] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, D. Hassabis, Mastering the game of Go with deep neural networks and tree search, Nature 529 (2016) 484–489.

[5] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, ..., D. Hassabis, Mastering chess and shogi by self-play with a general reinforcement learning algorithm, arXiv preprint arXiv:1712.01815 (2017).

[6] S. Ba, T. Hiraoka, T. Onishi, T. Nakata, Y. Tsuruoka, Monte Carlo Tree Search with variable simulation periods for continuously running tasks, 2019 IEEE 31st International Conference on Tools with Artificial Intelligence (2019) 416–423.

[7] R. Coulom, Efficient selectivity and backup operators in Monte-Carlo Tree Search, in: International conference on computers and games, volume 26, 2006, pp. 72–83.

[8] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Colton, A survey of Monte Carlo Tree Search methods, IEEE Transactions on Computational Intelligence and AI in games 4 (2012) 1–43.

[9] M. Świechowski, K. Godlewski, B. Sawicki, J. Mańdziuk, Monte Carlo Tree Search: A review of recent modifications and applications, Artificial Intelligence Review 56 (2023) 2497–2562.
[10] S. Ontanón, The combinatorial Multi-Armed Bandit problem and its application to real-time strategy games, in: In Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, volume 9, 2013, pp. 58–64.
[11] P. Šmejkal, J. Gemrot, Engaging turn-based combat in the Children of the Galaxy videogame, in: Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, volume 14, 2018.
[12] F. Frydenberg, K. R. Andersen, S. Risi, J. Togelius, Investigating MCTS modifications in general video game playing, IEEE Conference on Computational Intelligence and Games (2015) 107–113.
[13] D. J. Soemers, C. F. Sironi, T. Schuster, M. H. Winands, Enhancements for real-time Monte-Carlo Tree Search in general video game playing, IEEE Conference on Computational Intelligence and Games (2016) 1–8.
[14] M. J. Tak, M. Lanctot, M. H. Winands, Monte Carlo Tree Search variants for simultaneous move games, 2014 IEEE Conference on Computational Intelligence and Games (2014) 1–8.
[15] C. H. Hsueh, I. C. Wu, W. J. Tseng, S. J. Yen, J. C. Chen, An analysis for strength improvement of an MCTS-based program playing chinese dark chess, Theoretical Computer Science 644 (2016) 63–75.
[16] L. Kocsis, C. Szepesvári, Bandit based Monte-Carlo planning, in: European conference on machine learning, 2006, pp. 282–293.
[17] D. Tolpin, S. Shimony, MCTS based on simple regret, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 26, 2012, pp. 570–576.
[18] P. Auer, N. Cesa-Bianchi, P. Fischer, Finite-time analysis of the Multiarmed Bandit problem, Machine learning 47 (2002) 235–256.
[19] K. Shibahara, Y. Kotani, Combining final score with winning percentage by sigmoid function in Monte-Carlo simulations, in: 2008 IEEE Symposium On Computational Intelligence and Games, 2008, pp. 183–190.
[20] T. Pepels, M. J. Tak, M. Lanctot, M. H. Winands, Quality-based rewards for Monte-Carlo Tree Search simulations, in: ECAI, 2014, pp. 705–710.
[21] F. Xie, Z. Liu, Backpropagation modification in Monte-Carlo game tree search, in: 2009 Third International Symposium on Intelligent Information Technology Application, volume 2, 2009, pp. 125–128.
[22] S. Ontañón, Informed Monte Carlo Tree Search for real-time strategy games, in: 2016 IEEE Conference on Computational Intelligence and Games (CIG), 2016, pp. 1–8.
[23] D. Churchill, M. Buro, Portfolio Greedy Search and simulation for large-scale combat in StarCraft, in: Proceedings of the Conference on Computational Intelligence in Games, 2013, pp. 1–8.
[24] D. Churchill, M. Buro, Hierarchical Portfolio Search: Prismata's RobustAI architecture for games with large search spaces, in: In Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, volume 11, 2015, pp. 16–22.
[25] Z. Šustr, J. Sitera, M. Mulač, M. Ruda, D. Antoš, L. Hejtmánek, P. Holub, Z. Salvet, L. Matyska, Metacentrum, the czech virtualized ngi, In EGEE Technical Forum (2009).

Declaration on Generative AI

During the preparation of this work, the author(s) used Gemini in order to: Formatting assistance. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.