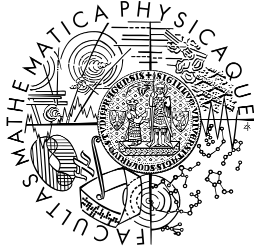


FACULTY OF MATHEMATICS AND PHYSICS
Charles University

MASTER THESIS

Peter Guba

Analysis of different MCTS implementations of artificial intelligence for the Children of the Galaxy computer game

KSVI

Supervisor of the master thesis: Mgr. Jakub Gemrot, Ph.D.

Study programme: Computer Graphics and Game Development

Study branch: IPGVPH

Prague 2022

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I would like to thank my supervisor, Jakub Gemrot, for his guidance and many useful insights.

Title: Analysis of different MCTS implementations of artificial intelligence for the Children of the Galaxy computer game

Author: Peter Guba

department: KSVI

Supervisor: Mgr. Jakub Gemrot, Ph.D., KSVI

Abstract: Monte Carlo Tree Search (MCTS) is a popular game AI algorithm that searches the state space of a game while using randomized playouts to evaluate new states. There have been many papers published about various adjustments of the original algorithm, however, work that compares multiple of these algorithms together does not seem to exist. This lack of data can make it difficult to decide which variant to use without implementing and testing them which is potentially quite time-consuming. The aim of this thesis is therefore twofold. First to create such a comparison in a specific setting and second to introduce a new variant, WP MCTS, which is based on the idea that one should be able to gather more information from a playout by taking a look at all the states encountered during its computation. For our setting, we chose battles between small armies in a 4X computer game called Children of the Galaxy. The results presented here indicate that many, though not all tested variants outperform basic MCTS in this setting.

Keywords: artificial intelligence, game tree, Monte Carlo methods, MCTS implementations, computer games, Children of the Galaxy, 4X games

Contents

1 Introduction **4**

2 Related Work **6**

3 MCTS **7**

3.1 Game tree 7
3.2 Monte Carlo methods 8
3.3 Multi-Armed Bandit 8
3.4 Algorithm 9
3.4.1 Selection 9
3.4.2 Expansion 9
3.4.3 Simulation 10
3.4.4 Backpropagation 10
3.5 Pros and Cons 11

4 Children of the Galaxy **12**

4.1 4X Games 12
4.2 Gameplay 13
4.3 Combat 13
4.4 Branching factor 14

5 Script-based search **16**

5.1 No-Overkill-Attack-Value (NOKAV) 16
5.2 Kiter 16

6 MCTS Variants **17**

6.1 UCT [9] 17
6.2 SR+CR MCTS [18] 17
6.3 VOI-aware MCTS [18] 19
6.4 UCB1-Tuned MCTS [1] 19
6.5 Sigmoid MCTS [15] 20
6.6 Relative Bonus MCTS [13] 21
6.7 Qualitative Bonus MCTS [13] 22
6.8 MCTS_HP [21] 22
6.9 FAP MCTS [19] 23
6.10 WP MCTS 24

7 Experiments **26**
7.1 Setup . . . 26
7.1.1 Definitions . . . 26
7.1.2 Test design . . . 26
7.1.3 Experiment Design . . . 27
7.1.4 Experiment Specification . . . 28
7.2 Abbreviations . . . 30
7.3 Results . . . 30
7.3.1 Playout Analysis . . . 30
7.3.2 Combat Analysis . . . 37

8 Conclusions and Future Work **49**
8.1 Conclusions . . . 49
8.2 Future Work . . . 50

Bibliography **51**

A User Documentation **53**
A.1 Data and Files . . . 53
A.2 XML Manager . . . 54
A.2.1 makeai . . . 54
A.2.2 makebmrk . . . 55
A.2.3 makebmrks . . . 56
A.2.4 makebset . . . 56
A.2.5 makebsets . . . 56
A.2.6 delete . . . 57
A.2.7 change . . . 57
A.2.8 sethome . . . 58
A.2.9 setscripts . . . 58
A.2.10 quit . . . 58
A.2.11 help . . . 58
A.3 CSV Cruncher . . . 59
A.4 Data Visualizer . . . 59
A.4.1 battledata1.py . . . 59
A.4.2 battledata2.py . . . 60
A.4.3 battlerankings.py . . . 60
A.4.4 common_functions.py . . . 60
A.4.5 playoutdata.py . . . 60
A.4.6 playoutrankings.py . . . 61
A.4.7 timedepthdata.py . . . 61

A.5 Position Generator 61
A.6 Result Trimmer 61
A.7 Time-Depth Cruncher 62

B Additional Data **63**

1 Introduction

Artificial intelligence (AI) for games is a very extensive field of research with a long history that arguably goes all the way back to the very first computer games. Problems from this field of study are interesting for two reasons. One is commercial – AI is often responsible for creating challenges that the player of a game has to overcome, such as controlling non-playable characters and the environment that the player interacts with, and is therefore often an essential part of the experience that a game provides. The other is scientific – games can often pose very hard problems for AI and these challenges have long been at the forefront of AI research with games such as chess [6] and Go [17] forming important milestones in this field.

Each game has its own AI requirements, most algorithms and approaches are therefore situation-specific and hard to generalise, but some algorithms for general game playing do exist nonetheless. Perhaps the most prominent of these is Monte Carlo Tree Search (MCTS). This algorithm partially explores the game tree of a game and uses randomized simulations to determine how good the encountered states are while trying to balance the exploration of new states and the exploitation of ones already deemed to be good. This approach is very versatile and able to combine with game-specific knowledge to create very skilful agents, such as AlphaGo, famously used by DeepMind to defeat the world Go champion for the first time.

Since the algorithm's introduction in 2006 [5], many adjustments have been proposed [3, 20], each claiming to outperform the original in some setting, however, we have not been able to find any paper that would compare any of these variants together. This is unfortunate 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 implementing and trying them which can potentially be quite time-consuming. In this thesis, we aim to take the first step towards remedying that by comparing ten different MCTS variants together. Nine of these are taken from papers by other authors while the ninth one, called WP MCTS, is our own variant which adjusts the way that playouts are evaluated.

It would of course be incredibly time consuming to gather data on how the variants compare in every possible context, so we chose only one – a 4X game called Children of the Galaxy (CotG).

4X is an abbreviation of Explore, Exploit, Expand, Exterminate and it is a term for a subgenre of strategy games. In general, these games involve creating some kind of empire through various means, from technological research to military expansion. 4X games can be divided into subtasks, each of which can be

managed by its own AI. In this thesis therefore, we are not building agents for the whole game but only for one such subtask – combat between units, which is a crucial part of CotG. We chose this context so that we can capitalize on a simulator created by Šmejkal [21]. Since we are working only with a small subtask of the game, the results obtained from this comparison should be generalizable to various games which involve strategic fights between groups of units.

The rest of this thesis is structured as follows: In the second section, we present some related work. In the third section, we go over how the original MCTS algorithm works. In the fourth section, we go over the workings of our testing environment, the Children of the Galaxy computer game. In the fifth section, we introduce script-based search - a restriction of traditional action-space search used to shrink the branching factor of our search. In the sixth section, we briefly go over the workings of the MCTS variants that we will be comparing. In the seventh section, we detail the experiments we have performed and their results. In the eighth and final section, we offer some conclusions and describe potential subjects for future work.

2 Related Work

The MCTS algorithm functions completely independently of any game-specific knowledge, although it does entail some assumptions about what kind of a game is being played - namely that it is a two-player discrete sequential game with perfect information. The algorithm can be tweaked however to accommodate deviations from these assumptions. This makes it very versatile and easily applicable to a wide variety of games.

Perhaps the most famous AI that utilizes MCTS is DeepMind's AlphaGo [17], which succeeded in defeating the world Go champion Lee Sedol in 2016. It accomplished this by combining the algorithm with two neural networks, one for evaluating game states and another which provided a policy for choosing moves.

Equally impressive is the fact that DeepMind later created an expanded AI called AlphaZero [16], capable of playing chess, shogi and Go. This AI defeated the world's strongest chess playing engine, Stockfish. This is especially interesting given the fact that chess is a game that is inherently difficult for MCTS to play due to the fact that states often have a very limited number of very good moves while all other moves are terrible.

MCTS has also been successfully used in agents that play other board games, such as Hex, where an agent called MoHex [7] which utilizes MCTS won the gold medal at the 2009 Computer Olympiad, and Arimaa [10], although it has not achieved top level performance among AIs there.

Notable results using this algorithm have likewise been made in the field of RTS games which is notoriously difficult for AIs to play due to factors like a large branching factor, imperfect information and their real-time nature. For example, the campaign AI of the game Total War: Rome II uses two MCTS implementations, one to control the allocation of resources to tasks and another to execute those tasks[1]. And in the field of scientific experimentation, Santiago Ontañón has obtained good results when combining MCTS with Bayesian models that estimate the probability distribution of actions of a strong player in the μRTS environment [11].

Agents based on MCTS have achieved good results in other real-time video games too, such as Ms. Pac-man [14].

And in the field of nondeterministic games, MCTS has achieved ntable success when it was utilised in poker to create the first exploiting no-limit Texas Hold'em bot that can play at a reasonable level in games of more than two players [2].

[1] https://web.archive.org/web/20170313041719/http://aigamedev.com/open/coverage/mcts-rome-ii/ [Accessed 15.07.2022]

3 MCTS

3.1 Game tree

Every video game has some state that can change through time. Playing the game means applying actions that change that state, i.e., move the game forward. The game must also offer the player multiple possible actions in a single state that can lead to different states, otherwise the player would just be linearly progressing through some predetermined set of states without any feeling of agency.

This structure can be expressed in the form of a tree with vertices corresponding to game states and directed edges corresponding to actions that lead from one state to another. The initial state of an edge is the one that is closer to the root.

Figure 1 shows an example of this data structure which is called a *game tree* and its exploration is an essential part of many game AI algorithms. If a computer is able to explore the entire game tree, it can play the given game perfectly as it always knows which of its actions lead to the best outcomes. This is rarely possible however, as many games have game trees that are far too large for a computer to explore under reasonable time and space constraints. The best one can hope for with such games is to explore only a part of the game tree in some organized fashion and use the gathered data to pick a good move.

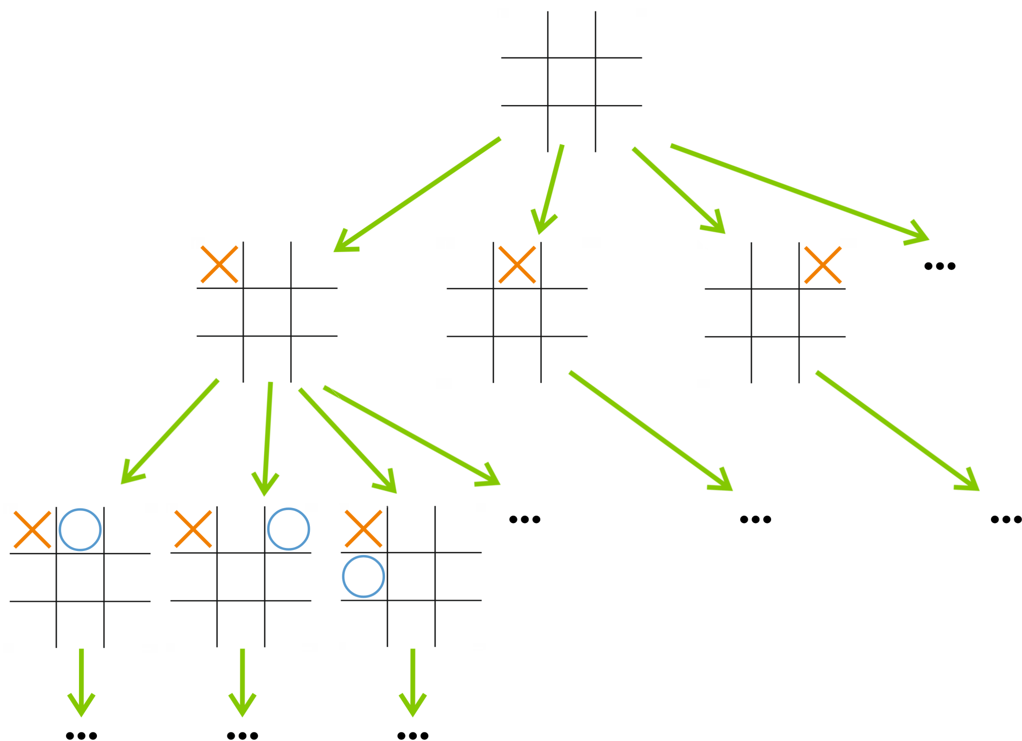


Figure 1: A partial game tree of the game tic-tac-toe.

Some important aspects of a game tree that we will be discussing in this work are its *depth* (the length of the longest path from the root to some leaf node) and *branching factor* (the average number of children that a non-leaf node has).

3.2 Monte Carlo methods

Monte Carlo methods are a broad class of algorithms that rely on randomly sampling some domain in order to get an estimate of some value. They are useful when it is difficult or impossible to compute the desired value exactly. A simple example is shown in figure 2 where a Monte Carlo based algorithm is used to approximate the area of a circle with a radius of 0.5.

The algorithm randomly samples a 1 by 1 square containing the circle N times. The circle's area is then approximated as $\frac{\#\text{points in the circle}}{\#\text{points outside the circle}}$.

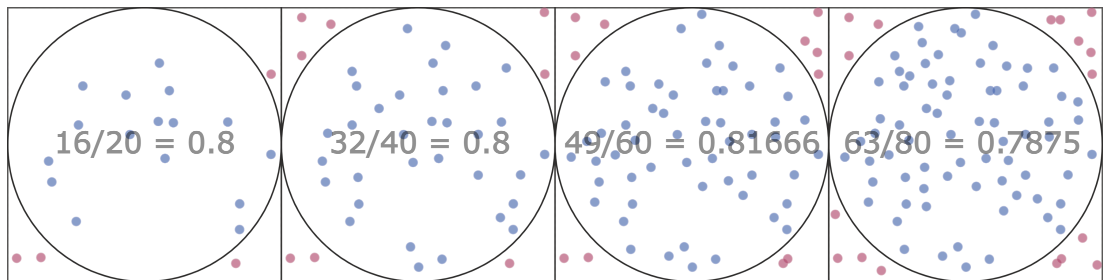


Figure 2: Monte Carlo method algorithm example.

3.3 Multi-Armed Bandit

The Multi-Armed Bandit (MAB) [1] is a well-known problem from probability theory and it is encountered repeatedly during the course of running MCTS, as will be explained in the following subsection.

In the most famous formulation of this problem, a gambler is faced with n slot machines (also known as one-armed bandits), each of which has an arm which can be pulled, after which he obtains a reward. The rewards associated with the arms have unknown probability distributions. The goal of the gambler is to gather as much reward as possible.

The MAB is a classic problem that showcases the exploration-exploitation trade-off dilemma - one must balance between allocating the arm pulls to exploration of arms that have not been pulled that much in order to explore how good they are and to exploitation of the arm that is believed to be the best.

3.4 Algorithm

MCTS is and algorithm that uses the Monte Carlo method to explore the game tree of a game. During its run time, it progressively builds a partial game tree where every node stores a game state, its child nodes, the number of times it has been visited and some score used to determine how good the state is for a player.

It is closely related to the MAB problem discussed previously - since it cannot explore the entire game tree, the algorithm must balance the exploration of new moves with the exploitation of ones already deemed to be good at every node. Every such decision can therefore be conceptualised as an instance of MAB.

The core of the algorithm consists of four steps – Selection, Expansion, Simulation and Backpropagation. These repeat in a loop until some terminal condition is reached (usually time limit being exceeded). The steps work as follows:

Algorithm 1 MCTS

```
1: function MCTS(state, algorithmBudget, playoutBudget)
2:     root ← MakeNode(state)
3:     while algorithmBudget not drained do
4:         nodeToExpand ← Select(root)
5:         newNode ← Expand(nodeToExpand)
6:         reward ← Simulate(newNode, playoutBudget)
7:         Backpropagate(newNode, reward)
8:     return root.GetBestChild()
```

3.4.1 Selection

In this step, the algorithm traverses the already explored part of the game tree, starting at the root. If it encounters a node that is not fully expanded yet, it moves on to the Expansion step. Otherwise, it decides which of the node's children to move to. The algorithm needs a strategy based on which to make this decision. This strategy is referred to as the *tree policy*. It is this policy that is responsible for balancing exploration and exploitation. We will be going over what such a policy can look like in section 6, where we discuss the different MCTS variants that we implemented.

3.4.2 Expansion

The algorithm picks a move which has not yet been tried in the state corresponding to the node picked in the previous step and creates a new node which holds the state to which that move leads. This new node is appended to the node picked in the previous step. The possible moves are not evaluated in any way and one is simply picked either at random or in some predetermined order.

Algorithm 2 MCTS Step 1: Selection

```
1: function SELECT(root)
2:     currentNode ← root
3:     bestNode ← null
4:     while bestNode is null do
5:         if currentNode not fully expanded then
6:             bestNode ← currentNode
7:         else
8:             currentNode ← GETBESTCHILD(currentNode, TREEPOLICY)
9:     return bestNode
```

Algorithm 3 MCTS Step 2: Expansion

```
1: function EXPAND(node)
2:     return currentNode.GETNEXTCHILD()
```

3.4.3 Simulation

A simulation (called a playout) of the game is performed, starting from the state corresponding to the newly created node. During this simulation, moves are picked at random. The strategy used to pick moves in this step is referred to as the *default policy* and it is usually different from the tree policy.

When the simulation ends (either because a terminal state is reached or because it is stopped by some other cut-off condition), the state at which it ended is evaluated and a numeric score is obtained.

Algorithm 4 MCTS Step 3: Simulation

```
1: function SIMULATE(startNode, budget)
2:     currentState ← startNode.STATE
3:     currentPlayer ← startNode.PLAYER
4:     while budget not drained and currentState is not terminal do
5:         nextMove ← GETMOVE(currentState, currentPlayer, defaultPolicy)
6:         currentState ← currentState.APPLYMOVE(nextMove)
7:         currentPlayer ← the other player
8:     return EVALUATE(currentState, startNode.PLAYER())
```

3.4.4 Backpropagation

The score obtained in the previous step is backpropagated up the tree and used to adjust the scores of all the nodes on the path from the newly created node to the root.

Algorithm 5 MCTS Step 4: Backpropagation

```
1: procedure BACKPROPAGATE(node, score)
2:     node.NUMOFVISITS + +
3:     node.SCORE += score
4:     if node.HASPARENT then
5:         BACKPROPAGATE(node.PARENT, -score)
```

3.5 Pros and Cons

The biggest advantages of MCTS are its simplicity, independence of game-specific knowledge, ability to stop at any time and asymmetric building of the game tree which allows it to adapt to various situations. These characteristics have allowed it to become a popular algorithm used in a wide variety of games.

To obtain good results however, this algorithm often needs to run a very large amount of iterations which can take a lot of time. Also, since it treats all possibilities as equal, it can have problems dealing with trap states (states which lead to immediate rewards but are bad in the long run) and making efficient sacrifices which doesn't make it a good fit for games like chess.

4 Children of the Galaxy

4.1 4X Games

Children of the Galaxy (CotG) belongs to the genre of 4X games which is a subgenre of strategy games. These games are usually very complex and require the player to balance various different aspects of gameplay, such as research, battles with enemies and gathering of resources. The goal is generally the creation of an empire using the steps denoted by the four Xs:

- Explore - The player has to explore new areas of the game world.
- Expand - The player expands his empire by building settlements
- Exploit - The player gathers resources from the areas he controls.
- Exterminate - The player destroys other players.

Figure 3: A screenshot of the galaxy view of CotG. The colored hexagons represent territory that is under the control of some player. The hexagons with light blue outlines contain units that are moving through interstellar space. The places that are marked with insignias of the different factions and have some stats displayed above them are the individual solar systems. In the upper right corner, a map of the entire galaxy can be seen. On the left side of the screen are the player's stats, a list of planets he controls and a menu bar from which the player can activate screens that control different parts of the game, like research and diplomacy.

4.2 Gameplay

CotG is a turn-based game played on a hexagonal grid that takes place in a single galaxy and revolves around colonizing solar systems. The player picks one of three available races and starts out with a single solar system with a single planet under his control. From there, he can expand by building space ships which can be used to discover and colonize new planets.

The player can also research technologies which can give him various advantages, such as new units, upgrades and more effective ways of gathering resources.

There are three possible winning conditions – colonizing over 25% of planets in the galaxy, colonizing all the home planets of other players and researching and building a Dyson sphere.

4.3 Combat

At some point, the player will encounter other races trying to expand their own empires whereupon a battle may ensue. A battle in CotG can involve more than two players, however, we decided to restrict ourselves to a two-player format as that is the setting that MCTS is meant to be used in.

Like the rest of the game, battles in CotG take place on a hexagonal grid and in turns. A turn consists of issuing commands to one's units.

A unit has the following important stats:

- *hull* - How much damage a unit can take. When this value becomes zero, the unit is destroyed.
- *damage* - How much damage the unit can deal in one turn.
- *shields* - Absorb part of damage that would otherwise be subtracted from the unit's hull when it is attacked.
- *power* - The amount of hexes that a unit can move in a single turn.
- *attack range* - How far a unit's attack can reach. If an enemy unit is outside this range, this unit cannot attack it.

In every turn, a player can issue commands to however many units he wants. A single unit can be issued multiple commands and the issuing of commands can be interleaved (for example, a player can first issue a command to unit u, then to unit v and then issue another command to unit u).

Since battles are not an isolated part of CotG, the player can issue many different commands to units, some of which do not necessarily have anything to

do with combat. For example, the player can order a unit to destroy itself, or to make it move to interstellar space, thereby removing it from the solar system where the battle is taking place. When a unit has gained enough experience, the player can also upgrade it.

From our perspective however, there are only two relevant commands - *Attack* and *Move*. Attack makes a unit deal damage to another unit, provided that it is within its attack range. Move makes the unit move to a given hexagon, given that it has enough power.

There is no time limit for a turn, it ends when the player decides to end it by clicking the "End Turn" button. As the battles are not an isolated part of the game, they do not have clearly defined beginnings or ends from the perspective of the game, but a battle can be thought to be over when one player has either retreated or all of his units were destroyed.

Figure 4: A screenshot of a small battle in CotG. The orange-red units belong to one player and the grey-blue ones to another. The white numbers represent experience that some units gained after destroying one of the enemy's units. The red and blue numbers represent the damage sustained to a unit and to a unit's shields respectively. In the bottom right corner of the screen is the End Turn button which allows the player to finish his turn.

4.4 Branching factor

Strategy games where players battle using a potentially large number of units which can move freely are notorious for their large branching factors which make them really hard for AIs to conquer [12] and CotG is no different in this aspect.

Let us take a look at a single unit with power p. The amount of hexagons this unit can move to is given by the formula $hCount(u) = 3 \cdot p \cdot (p+1) = 3p^2 + 3p$. For a unit with power 4, this gives us $3 \cdot 4^2 + 3 \cdot 4 = 60$ possible hexagons, so if we only consider the possible moves of one such unit, we have a branching factor of 60.

For simplicity, let us dispense with the fact that unit actions can be interleaved and suppose that all units must perform their actions at once. If we then have 5 units with power 4, we already have a branching factor of $60^5 \approx 7,78 * 10^8$. This is without even taking into account the attack actions.

Players in strategy games often manipulate units which number in at least lower tens. It can therefore be easily observed that such a setting produces a branching factor that is absolutely intractable for ordinary consumer hardware to work with.

There are various approaches to this problem. A common one is dispensing with search altogether and instead creating an AI solely based on scripts. The disadvantage of this approach is that the AI will forever be stuck in the same patterns which the player can learn and then find ways to counter.

Another approach lies in using search techniques that do not rely on building a tree. Notable examples which have achieved good results are Portfolio Greedy Search [4] and Online Evolution [8].

The previous two approaches overcome the problem of a large branching factor by getting rid of the necessity to explore a substantial portion of the game tree. If one wants to stick to an AI based on tree search, it is necessary to somehow restrict the branching factor. This can be in principle be done using one of two approaches - *clustering* and *script-based search*.

Clustering means that the AI separates units into groups and assigns the same action to every unit. If we imagine for example that we have a CotG game where an AI controls 30 units with power 4, this gives us a branching factor of around $60^{30} \approx 2,21 \cdot 10^{53}$. If the AI employs clustering to split the units into 5 groups of 6 units, the branching factor is reduced to $60^5 \approx 7,78 \cdot 10^8$ which is a reduction by a factor of 10^{45}.

Script-based search, on the other hand, relies on restricting the actions available to units. As this is the technique we chose to use, we offer a detailed explanation of its workings in the following section. The reason we decided to use this technique was that it was already implemented in the software created by Šmejkal [21] on top of which we were building.

5 Script-based search

By default, searching for a good action for a unit to take is done in the space of all possible actions. In script-based search however, the search is done in *script space* - a subset of actions from action space that is offered by scripts. A script in this context can be conceptualised as a function which takes the current game state and a unit as input and outputs a possible action for that unit (it can also output multiple actions, but in our work, we restrict ourselves to scripts that return a single action).

Exploring only actions offered by a few scripts greatly reduces the branching factor. If we again consider the example where a player has 30 units with power 4 and consider using script-based search with 2 scripts, the branching factor is reduced to approximately $2^{30} \approx 1,07 \cdot 10^9$ which is a reduction by a factor of 10^{44}.

This decrease comes at an increased computational cost, as each script needs to perform some sort of computation on the current game state. It is therefore necessary to choose scripts which are easily computable.

We chose scripts No-Overkill-Attack-Value (NOKAV) and Kiter as these are commonly used for creating these kinds of action abstractions [4] and were already present in the framework created by Pavel Šmejkal. They function as follows:

5.1 No-Overkill-Attack-Value (NOKAV)

Find and attack an enemy unit u in weapons range with the highest value of $damage(u)/hp(u)$ where $damage(u)$ is the amount of damage the unit can deal and $hp(u)$ is the amount of hit points the unit has left. If the unit has already been assigned a lethal amount of damage by another unit however, ignore it and try to attack the unit with next highest $damage(u)/hp(u)$ (while applying the same condition). If there are no enemy units in range that fit the criteria, move towards the closest enemy unit.

In CotG, the value that strictly corresponds to the commonly used term hit points is the hull of a unit. However, since a unit also has shields, using NOKAV with only this value would be incorrect, since it would expect a unit to be dealt a lethal amount of damage, when the unit would in fact survive. We are therefore using the sum of hull and shields as hit points in the NOKAV implementation.

5.2 Kiter

Like NOKAV, but the unit also moves away from the enemy after having attacked.

6 MCTS Variants

In our search for MCTS variants to implement, we went over a number of papers, including an extensive survey of MCTS techniques [3]. Only a few of the variants we encountered turned out to be actually applicable to our testing environment however, therefore this was the only criterion based on which we chose which of them to implement. The following is a list of the variants we have implemented.

6.1 UCT [9]

Upper Confidence Bounds applied to Trees (UCT) is the most well-known variant of MCTS. It balances the exploration of new states and the exploitation of ones already considered to be good using the UCB1 formula [1].

UCB stands for Upper Confidence Bounds and it is a formula that was developed for the MAB problem. It looks as follows:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln n}{n_i}}$$

w_i is the number of wins associated with the node, n_i is the number of times the considered node has been visited, n is the number of times its parent has been visited and c is the exploration parameter, which is usually set to $\sqrt{2}$.

The default policy of this variant simply consists of randomly picking moves and it evaluates the final states of playouts to 1, 0, or -1, depending on whether the result was a win (or closer to a win, if the state was not terminal), draw or loss (or closer to a loss, if the state was not terminal). As we are working in script space, the random moves it picks are picked from the set of moves offered by the used scripts.

Note that the default policy and way of evaluating final states used here is not a part of the specification of this variant – UCT only defines the tree policy. We decided to go with these two approaches because they can be considered the most basic implementations of these two parts of the MCTS algorithm, just like UCT can be considered the most basic implementation of the tree policy (not because of its simplicity but because of how widely it is used).

The approaches used in this variant are taken as defaults, so unless a variant is mentioned explicitly to differ in some aspect from UCT, it does not.

6.2 SR+CR MCTS [18]

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

Algorithm 6 UCT Selection

```
1: function SELECT(root)
2:     currentNode ← root
3:     bestNode ← null
4:     while bestNode is null do
5:         if currentNode not fully expanded then
6:             bestNode ← currentNode
7:         else
8:             currentNode ← GETBESTCHILD(currentNode, UCB1)
9:     return bestNode
```

Algorithm 7 UCT Evaluation

```
1: function EVALUATE(state, player)
2:     if state is a draw then
3:         return 0
4:     else
5:         if state.WINNER = player or player is in a better position then
6:             return 1
7:         else
8:             return -1
```

for an optimal decision and the reward for a decision made based on some strategy. Simple regret is measured over a single decision while cumulative regret is measured over multiple decisions.

The UCB1 formula that is used by UCT aims to minimize the cumulative regret of all the arm pulls in an MAB setting. The algorithm only gets a reward for actually performing a move at the end of its computation however, so UCB1 causes the algorithm to exploit the best moves a lot more than it needs to.

In order to remedy this issue, this variant uses a policy that minimizes simple regret at the root (otherwise, it uses UCB1).

Algorithm 8 SR+CR Selection

```
1: function SELECT(root)
2:     currentNode ← root
3:     bestNode ← null
4:     while bestNode is null do
5:         if currentNode not fully expanded then
6:             bestNode ← currentNode
7:         else
8:             if currentNode == root then
9:                 currentNode ← GETBESTCHILD(currentNode, rootPolicy)
10:            else
11:                currentNode ← GETBESTCHILD(currentNode, UCB1)
12:    return bestNode
```

6.3 VOI-aware MCTS [18]

Like SR+CR MCTS, this variant tries to minimize simple regret when picking actions at the root. It does this by approximating the value of information (VOI) provided by the playouts using the myopic assumption that the algorithm will only sample one of the available actions. It then selects the action with the highest estimated VOI.

The formulas for the VOI approximation look as follows:

$$VOI_\alpha \approx \frac{\overline{X}_\beta}{n_\alpha + 1} \exp(-2(\overline{X}_\alpha - \overline{X}_\beta)^2 n_\alpha)$$

$$VOI_i \approx \frac{1 - \overline{X}_\alpha}{n_i + 1} \exp(-2(\overline{X}_\alpha - \overline{X}_i)^2 n_i), \quad i \neq \alpha$$

where $\overline{X}_i$ is the average reward for action i, n_i is the number of times the action was tried, $\alpha = \arg\min_i \overline{X}_i$ and $\beta = \arg\min_{i, i \neq \alpha} \overline{X}_i$.

To get a more in-depth understanding of how these formulas work and how they were derived, we encourage the reader to take a look at the original paper.

The pseudocode of the selection step for this variant is the same as for the SR+CR variant.

6.4 UCB1-Tuned MCTS [1]

UCB1-Tuned is a formula that improves on the original UCB1 formula in that it more tightly bounds the uncertainty of our observations. The formula is defined as follows:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln n}{n_i} \cdot \min\{1, V_i\}}$$

$$V_i = \sigma + \sqrt{\frac{2 \ln n}{n_i}}$$

where σ is the current sample variance.

The uncertainty of observations is multiplied here by an upper confidence bound of the variance. The constant 1 is derived from the fact that the back-propagated scores can take on the values -1, 0 and 1 which means that 1 is the maximum possible variance. If the score has a different possible range of values, this constant needs to be adjusted accordingly.

This variant uses this formula as the tree policy.

Algorithm 9 UCB1-Tuned Selection

```
1: function SELECT(root)
2:     currentNode ← root
3:     bestNode ← null
4:     while bestNode is null do
5:         if currentNode not fully expanded then
6:             bestNode ← currentNode
7:         else
8:             currentNode ← GETBESTCHILD(currentNode, UCB1TUNED)
9:     return bestNode
```

6.5 Sigmoid MCTS [15]

In the discussion of UCT, we mentioned that the most basic way of evaluating states is assigning them a score of 1, 0 or -1 based on which player won. This is called the win-or-lose approach. Another basic approach is using some sort of final score which evaluates how good the state is in more detail. For example, in a game of chess, one might compute the difference between the sums of values of pieces belonging to the two teams. This is called the final score approach.

Both of these approaches can work well under different circumstances. Final score works better when the number of simulations is higher, as it captures more information than win-or-lose but also suffers from a much higher variance. Win-or-lose therefore performs better when the number of simulations is lower, however, because it makes the algorithm only care about winning or losing, it can make it play safe moves when it is in a favourable position and play risky moves otherwise, causing it to win by a neck and lose big.

As both of these approaches offer different advantages the authors of this variant decided to try and combine them. They did so by making an algorithm which computes the final score, to which it then applies the sigmoid function:

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

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

Final score in our case is computed as the difference between the sums of remaining hull of all the units of the two teams. If the state is terminal, it amounts to the sum of the remaining hull of the units belonging to the winning player.

Algorithm 10 Sigmoid Evaluation

1: **function** EVALUATE(*state*, *player*)
2: $score \leftarrow state.\text{GETUNITHULLDIFFERENCE}(player)$
3: **return** $\frac{1}{1+\exp^{-k \cdot score}}$

6.6 Relative Bonus MCTS [13]

This variant applies a variance reduction method called *control variates* to the evaluation of a playout. This technique takes advantage of the correlation between two random variables, provided that one of them has a known mean, in order to create an unbiased estimator with a reduced variance. If we consider X to be an unbiased estimator of a value we want to compute and Y to be a random variable with a known mean, we can compute a new value, Z, in the following way:

$$Z = X + a \cdot (Y - \mathbb{E}(Y))$$

We can see that Z is also an unbiased estimator of $\mathbb{E}(X)$. If X and Y have a non-zero correlation, it is provable that there exists a value $a^* = -\text{Cov}(X, Y)/\text{Var}(Y)$ which minimizes $\text{Var}(Z)$.

In our case, X is set to be the evaluation of the playout and Y is a bonus computed based on the length of the playout and the depth at which it starts. The idea is that the longer the playout, the less reliable the information obtained from it.

The bonus is computed as follows. First, a standardized value λ is computed from an online approximation of the mean ($\bar{D}^{\tau}$) and standard deviation ($\hat{\sigma}_D^{\tau}$) of the playout lengths (the τ index denotes the player for whom the value is computed).

$$\lambda = \frac{\bar{D}^{\tau} - d}{\hat{\sigma}_D^{\tau}}$$

Then, this value is passed through a sigmoid function in order to bound and shape the values of the bonus.

$$b(\lambda) = (-1 + \frac{2}{1 + e^{-k\lambda}})$$

The parameter k is a constant to be determined by experimentation. Finally, this value is multiplied by α and sign of the reward r obtained from the playout.

$$r' = r + sgn(r) \cdot \alpha \cdot b(\lambda)$$

α is approximated as $|\widehat{\text{Cov}}(\overline{Y}^w, \overline{Y})/\widehat{\text{Var}}(\overline{Y})|$, where Y is the length of a playout

and Y^w is another random variable such that Y_i^w equals Y_i if player w won the playout and 0 otherwise. The value Y_i^w is not computed for each player separately, like with $\bar{D}^\tau$ and $\hat{\sigma}_D^\tau$, but instead one player is chosen and the value is always computed with respect to that player.

Algorithm 11 Relative Bonus Evaluation

```
 1: function EVALUATE(state, player)
 2:     if state is a draw then
 3:         score ← 0
 4:     else
 5:         if state.WINNER = player or state is more beneficial for player then
 6:             score ← 1
 7:         else
 8:             score ← −1
 9:     λ ← (D̄^τ − d) / σ̂_D^τ
10:     bonus ← sgn(score) · α · (−1 + 2/(1+e^(−kλ)))
11:     return score + bonus
```

6.7 Qualitative Bonus MCTS [13]

Just like Relative Bonus MCTS, this variant adds a bonus to the score that is computed at the end of every playout. This time however, the bonus is computed based on the quality of the last state of the playout, which in our case is the difference between the sums of hull of all the units of the two teams (the same metric which we used in the Sigmoid MCTS variant as the final score metric). The formulas and pseudocode are the same as in the previous variant, except for the way the lambda value is computed.

$$\lambda = \frac{q - \bar{Q}^\tau}{\hat{\sigma}_Q^\tau}$$

The order of subtraction in the computation is switched because a higher q value here means that the state is better, while in the previous variant, a lower d value meant that the playout was better.

We also tried combining the two bonuses, but we did not find this combination produced improved results.

6.8 MCTS_HP [21]

Instead of just backpropagating whether a player won or lost in a playout, this variant backpropagates the difference of the remaining hull of the two teams. This is again the same metric which we used in the Sigmoid MCTS variant as the

final score part of the computation and its purpose is to provide more detailed information about the quality of the final state to the algorithm.

Instead of just adding up these scores at every node when backpropagating however, this variant also normalizes them by the sum of all the hull of units of the winning player at the given state. This is done to preserve the information of how much of a loss the player suffered. For example, if the player retains two units with full hull in the final state of the playout, the quality of that result is different if he started with three units compared to if he started with twenty.

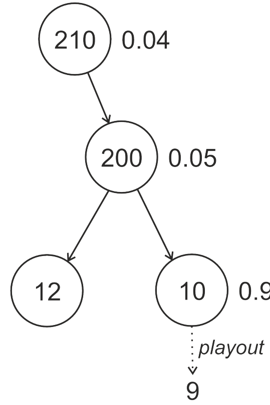


Figure 5: An example of how MCTS_HP backpropagation works. The numbers in the nodes are the sums of the hull of the remaining units of a player, the numbers next to them are the normalised values obtained from the result of the playout.

Algorithm 12 MCTS_HP Evaluation

1: **function** EVALUATE(*state*, *player*)
2:     **return** *state*.GETUNITHULLDIFFERENCE(*player*)

Algorithm 13 MCTS_HP Backpropagation

1: **procedure** BACKPROPAGATE(*node*, *score*, *winner*)
2:     *node*.NUMOFVISITS + +
3:     $node.\text{SCORE} \leftarrow node.\text{SCORE} + \frac{score}{\text{SUMOFHULL}(winner)}$
4:     **if** *node*.HASPARENT **then**
5:         BACKPROPAGATE(*node*.PARENT, -*score*, *winner*)

6.9 FAP MCTS [19]

This variant partitions playouts into groups based on the time at which they are computed (the later the better) and assigns a multiplicative factor to each group. Every playout from a group is considered to be worth n playouts, where n is the multiplicative factor assigned to the group. The reason the playouts

are partitioned this way is because of the assumption that playouts that are performed later offer more information.

The authors suggested two schemes for partitioning – linear and exponential. In the former, the playouts are partitioned into K equal parts. In the latter, the size of segments increases exponentially, so the n-th segment has a size of $S \cdot (2^{n-1}/2^K)$, where S is the total number of simulations.

The authors similarly suggested a linear and exponential scheme for computing the multiplicative factor. In the former scheme, the multiplicative factor is just the number of the segment, whereas in the latter, it is 2^{n-1}.

It is important to note that, unlike the other variants, this one relies on knowing the number of playouts beforehand. This information is usually not available however and the algorithm's execution is instead given a time limit. It should be possible to adapt this variant to such a setting by either setting the expected number of playouts to some number and doing computations based on that, or by computing the weight of a playout based on the time at which it occurs, but it is not certain whether such an adaptation would result in a variant with performance similar to this one.

Algorithm 14 FAP MCTS Evaluation

```
1: function Evaluate(state, player, currentIterationNum, maxIterations)
2:     n ← GetSegment(currentIterationNum, maxIterations)
3:     multiplicativeFactor ← 2^(n−1)
4:     if state is a draw then
5:         return 0
6:     else
7:         if state.Winner = player or state is more beneficial for player then
8:             return multiplicativeFactor
9:         else
10:            return -multiplicativeFactor
```

6.10 WP MCTS

Weighted Propagation (WP) MCTS is our own variant. It is based on the idea that, given that one has a good heuristic available for measuring state quality, it should be possible to extract more useful information out of a playout than just what outcome it leads to by simply taking into account how the states encountered in the playout evolved through time.

In order to achieve this, we have tried various functions which assigned weights to the information in the different states in the playout, trying to take into account both the fact that states the occur later have a lower probability of being encountered than those that occur earlier and the fact that states occur earlier

offer less information about how good the current state is than states that occur later. Out of the approaches that we tried however, the best one turned out to be a simple average of the scores of all the states encountered during the playout.

How well this variant performs equipped with a different function is certainly a potential topic for future research.

For our heuristic function, we chose the same function that was used in variants MCTS_HP, Qualitative Bonus MCTS and Sigmoid MCTS to judge the quality of a state - the difference between the sums of hulls of the remaining units of the two teams.

Algorithm 15 WP MCTS Evaluation

```
1: function EVALUATE(states, player)
2:     score ← 0
3:     for all state ∈ states do
4:         stateScore ← state.GETUNITHULLDIFFERENCE(player)
5:         pos ← state.POSITIONINPLAYOUT
6:         length ← states.LENGTH
7:         score ← score + IMPORTANCEFUNCTION(stateScore, pos, length)
8:     return score
```

7 Experiments

In this section, we go over how we designed our experiments, what data we obtained and what they might mean. The data presented here are always averaged over a number of battles selected according to criteria which will be explained later. More detailed data on the individual metrics we measured can be found in section B of the appendix.

Our aim was to determine whether some of the MCTS variants that we discussed in section 6 could be said to perform better than others. In order to do this, we created tests in which we pitted the variants against each other and ran them in a round-robin tournament fashion while gathering data about the performance of the different variants.

7.1 Setup

7.1.1 Definitions

- variant - one of the algorithms described in section 6
- combat setting - the number, types and positions of units used in a set of battles
- playout setting - the number of playouts available to an MCTS variant
- algorithm setting - the settings of parameters of a specific MCTS variant
- test - a set of 1 vs 1 battles during which data is gathered

7.1.2 Test design

Each test that we ran was parameterized by two variants with specific algorithm settings and a playout setting which was identical for both variants. The reason why we decided to impose a restriction on playouts rather than time was that such results are independent of the underlying software (the operating system and its configuration) and hardware that they are run on. Ten years from now, any paper published now about MCTS that restricts how long the algorithm can run will be outdated and its results no longer applicable. Results based on restricting the number of playouts however will always remain replicable and relevant. We also hypothesize that restricting the number of playouts should ensure that the agents perform the same on every computer.

Each test consisted of individual battles that were separated into categories based on the number of units that each team had at its disposal and multiple battles were conducted in each category in order to get more reliable results.

For each of the two variants participating in a test, we measured six statistics - the number wins, the number of symwins, the remaining hull of the units of the winning player, the amount of damage dealt to the other player, iteration time and tree depth, with the last two being measured on a separate set of tests.

A symwin is achieved either when an agent wins the same battle both when it goes first and when it does not, or when it wins only one battle but its remaining hull at the end of the two battles is higher. If one agent wins each battle and they have the same amount of hull left at the ends of the two battles, neither of them gets a symwin (such a result is called a symdraw).

The reason why we place emphasis on remaining hull is that a strategy game is usually not about winning a single battle but about winning multiple battles in order to accomplish some larger goal. Units are therefore reused between battles and how many units are preserved after a battle and what condition they are in is important. These two pieces of information are reflected in the remaining hull at the end of a battle.

We will refer to the first four metrics (wins, symwins, hull and damage) as primary metrics and to the remaining two (time and depth) as secondary metrics.

7.1.3 Experiment Design

It would have been ideal if we had tested all possible pairs of MCTS variants with multiple possible settings in many different scenarios with both large and small numbers of units. It was clear from the start however, that due to time constraints this would not be feasible, so we had to narrow down our domain of experimentation.

We started with testing all the possible algorithm settings we wanted to try. We then tested each of these against UCT which, as we mentioned in section 6.1, can be considered to be the default implementation of MCTS. Based on the results, we picked the best performing algorithm settings (with at least one setting for each variant) and tested every possible pair. We then selected one algorithm setting for each variant and ran one more round of tests, the results of which are presented in this paper.

We ran one more round of tests where we pitted each variant (with algorithm settings picked in the previous rounds) against UCT once again in order to measure the average iteration time and average tree depth. The reason for this was that the other tests were run in parallel in order to produce results as fast as possible. However, we did not find a tool that would allow us to accurately measure the time of each variant while they were run in parallel, so to measure this metric, we had to run tests sequentially. In order to do this in a reasonable

amount of time, we restricted the tests to only fights against UCT.

7.1.4 Experiment Specification

Listing all the possible algorithm settings we wanted to try yielded 70 different settings files. From the first round of tests where all the variants with all their different algorithm settings were run against UCT (with the exception of UCT itself), we picked 27 algorithm settings that performed the best with at least one setting belonging to each variant. In the second round, we tested these algorithm settings against each other. After this, we picked one algorithm setting for every variant which we moved to the final round of tests. This meant $69+351+45 = 465$ tests in total, plus the 10 sequential tests to measure iteration time and depth. The settings we describe in the rest of this section were applied only to the final round of tests, while the preliminary rounds ran under simplified conditions in order to ensure fast execution.

The chosen algorithm settings for the final round were the following:

- FAP MCTS - we set the number of segments to 100 and picked the exponential segmentation scheme and linear multiplication scheme.
- Relative Bonus MCTS and Qualitative Bonus MCTS - we set the value of the parameter k to 0.1.
- Sigmoid MCTS - the parameter k was also set to 0.1.
- SR+CR MCTS - the simple-regret-maximizing policy that we chose was ϵ-greedy. This policy picks the current best move with probability ϵ and picks any other move with probability $\frac{1-\epsilon}{N-1}$, where N is the number of moves. The parameter ϵ was set to 0.75.

The other variants - UCT, MCTS_HP, VOI-aware MCTS, UCB1-Tuned MCTS and WP MCTS did not have any parameters to set. All of our experiments were conducted on a computer with AMD Ryzen Threadripper 1950X 16-Core CPU @ 3.39Ghz and 32GB of RAM. The operating system was Windows 7 Professional and the software was written in C# targeting .NET version 4.7.2.

In our battles, we used two types of units - *Destroyers* and *Battleships*. The former is a unit with high damage but short attack range and low hull while the latter is a unit with medium attack range, lower damage, but high hull and shields. As for the distance which they can move in a single turn, a Destroyer has a movement radius of 5 while a Battleship has a movement radius of 6. Both of these units were used without any upgrades. While the game offers other units, we felt that these two provided enough diversity for our purposes.

In every test in the final round we used 6 combat settings with 6 battles played on each setting. The settings were symmetric with respect to the number of units and their positioning (an example of such positioning can be seen in Figure 6). The player with the first move alternated between the battles, so in half the battles one side had the first move and in half the battles the other side. A battle was considered finished when only one player remained. There was a round limit of 1000 set for the battles, the exceeding of which would have caused the battle to terminate and the winner to be decided based on the remaining hull, but this limit was never reached (the maximum number of rounds reached in any battle was 139 and the average was around 52).

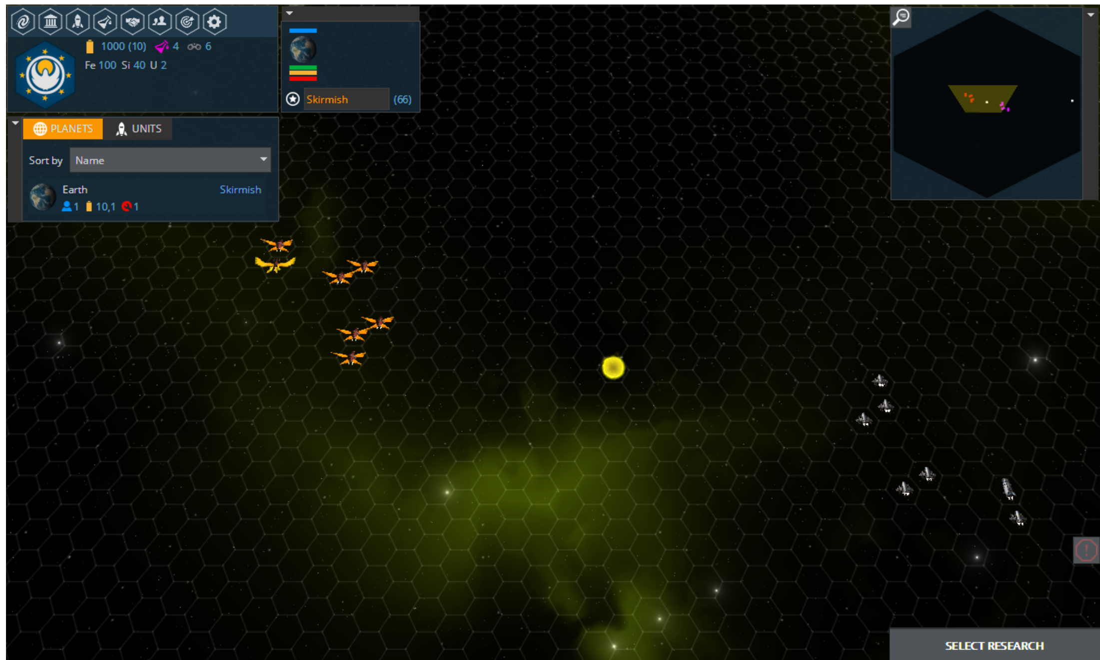


Figure 6: An example of a symmetric combat setting. Both teams have seven units, the positions of which are mirrored through the origin.

The 6 combat settings that be used can be divided into three pairs - 4 vs 4 and 8 vs 8, which represent small scale combat, 16 vs 16 and 32 vs 32, which represent medium scale combat and 48 vs 48 and 64 vs 64, which represent large scale combat. In all settings, half the units were battleships and half were destroyers. The positions for units of one team for each setting were randomly generated beforehand and remained constant. As the battles were symmetric, this determined the positions for the other team too. Since the positions and unit numbers are uniquely determined by the number of units on each team, we will denote different combat settings by team sizes (like 4 vs 4) for brevity.

We also chose 3 playout settings - 100, 500 and 1000. This may seem like too few, as it is not unusual for MCTS to get to run tens of thousands or even a million iterations in some games. However, in CotG, running even 100 playouts

on a consumer PC took a number of seconds, therefore we find these numbers to be sufficient.

7.2 Abbreviations

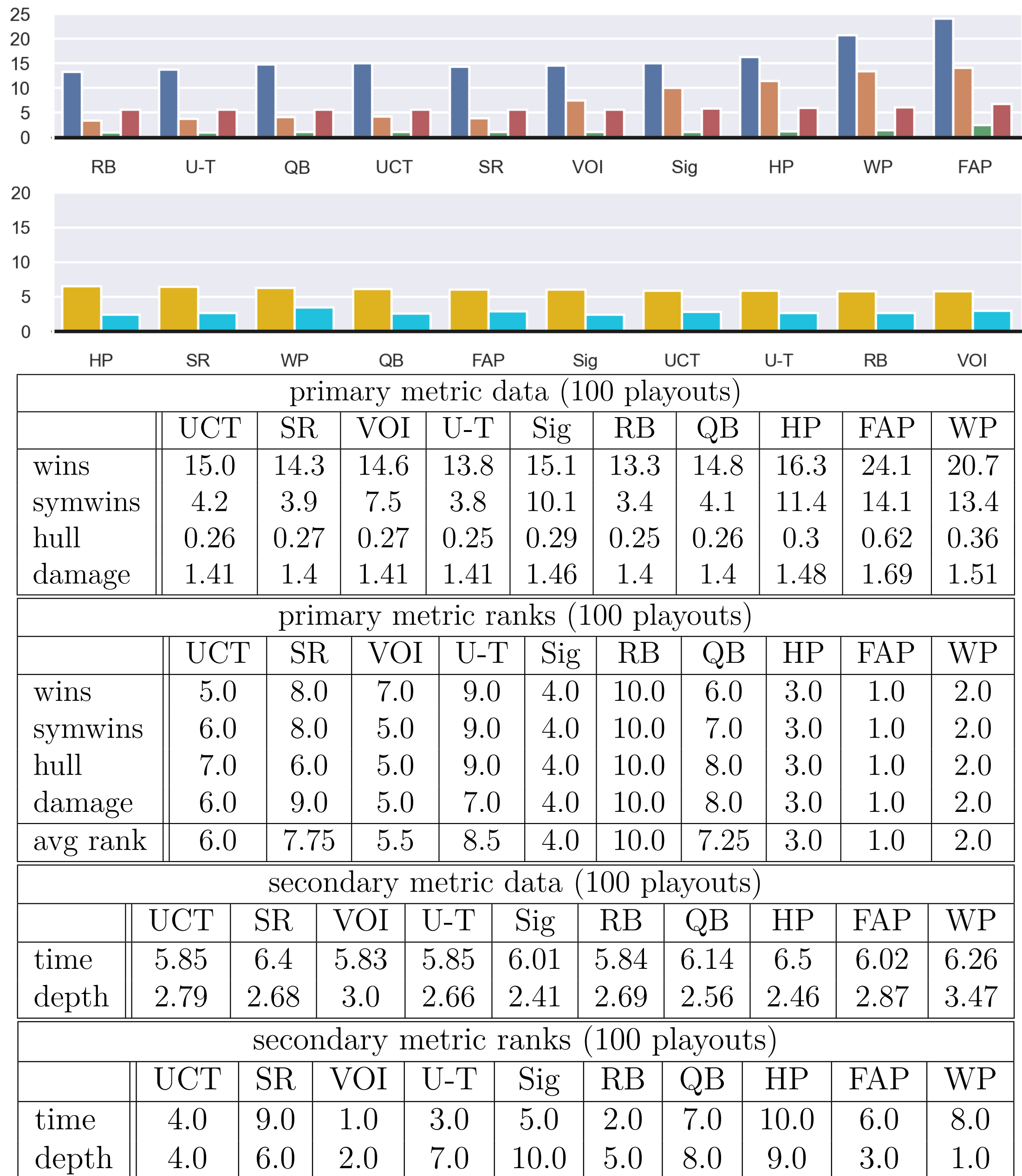
In the following subsection, we present our results. These results include tables and graphs that visualise our data. For the sake of readability, we needed to abbreviate the names of the tested variants. We list these abbreviations here.

- SR+CR MCTS - SR
- VOI-aware MCTS - VOI
- UCB1-Tuned MCTS - U-T
- Sigmoid MCTS - Sig
- Relative Bonus MCTS - RB
- Qualitative Bonus MCTS - QB
- MCTS_HP - HP
- FAP MCTS - FAP
- WP MCTS - WP

7.3 Results

7.3.1 Playout Analysis

In this section, we take a look at data on each of the variants that have been accumulated based on the number of playouts that the variant had at its disposal. All the data presented here are averaged - wins, symwins, depth and iteration time are averaged over the number of battles, hull and damage over the total number of units at an agent's disposal across all battles. The graphs with primary metrics in figures 7, 8, 9 and 10 are sorted according to the remaining hull from left to right in ascending order. The graphs with secondary metrics in those figures are sorted according to average iteration time. This metric is different from others in that less means better, so the variants are sorted from left to right in descending order. The variants in the tables are presented in the order in which they were introduced. The values in the tables are rounded up to two decimal places which can sometimes make it seem like two variants performed equally well on some metric, when in reality they achieved a score which differed by too small an



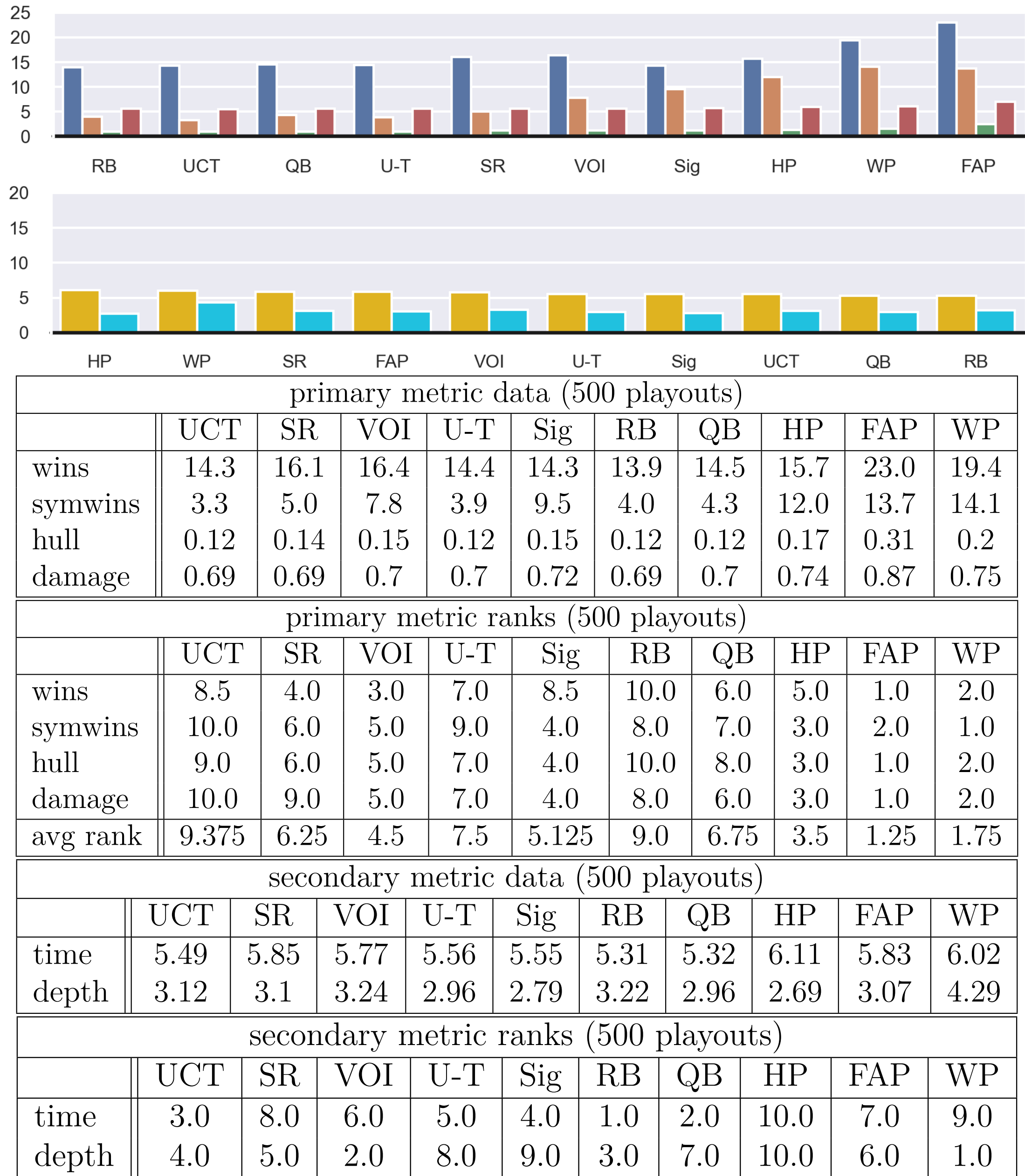
primary metric data (100 playouts)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	15.0	14.3	14.6	13.8	15.1	13.3	14.8	16.3	24.1	20.7
symwins	4.2	3.9	7.5	3.8	10.1	3.4	4.1	11.4	14.1	13.4
hull	0.26	0.27	0.27	0.25	0.29	0.25	0.26	0.3	0.62	0.36
damage	1.41	1.4	1.41	1.41	1.46	1.4	1.4	1.48	1.69	1.51

primary metric ranks (100 playouts)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	5.0	8.0	7.0	9.0	4.0	10.0	6.0	3.0	1.0	2.0
symwins	6.0	8.0	5.0	9.0	4.0	10.0	7.0	3.0	1.0	2.0
hull	7.0	6.0	5.0	9.0	4.0	10.0	8.0	3.0	1.0	2.0
damage	6.0	9.0	5.0	7.0	4.0	10.0	8.0	3.0	1.0	2.0
avg rank	6.0	7.75	5.5	8.5	4.0	10.0	7.25	3.0	1.0	2.0

secondary metric data (100 playouts)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	5.85	6.4	5.83	5.85	6.01	5.84	6.14	6.5	6.02	6.26
depth	2.79	2.68	3.0	2.66	2.41	2.69	2.56	2.46	2.87	3.47

secondary metric ranks (100 playouts)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	4.0	9.0	1.0	3.0	5.0	2.0	7.0	10.0	6.0	8.0
depth	4.0	6.0	2.0	7.0	10.0	5.0	8.0	9.0	3.0	1.0

Figure 7: Data accumulated over tests with the number of playouts set to 100. The first graph shows the average number of wins (blue), symwins (orange), the average remaining hull (orange) and damage dealt to the enemy (red). It is sorted according to the remaining hull. The second graph shows the average iteration time (yellow) and tree depth (cyan). It is sorted according to the former metric. The four tables report specific values of the data and the ranks of the different variants when they are sorted according to a given metric.



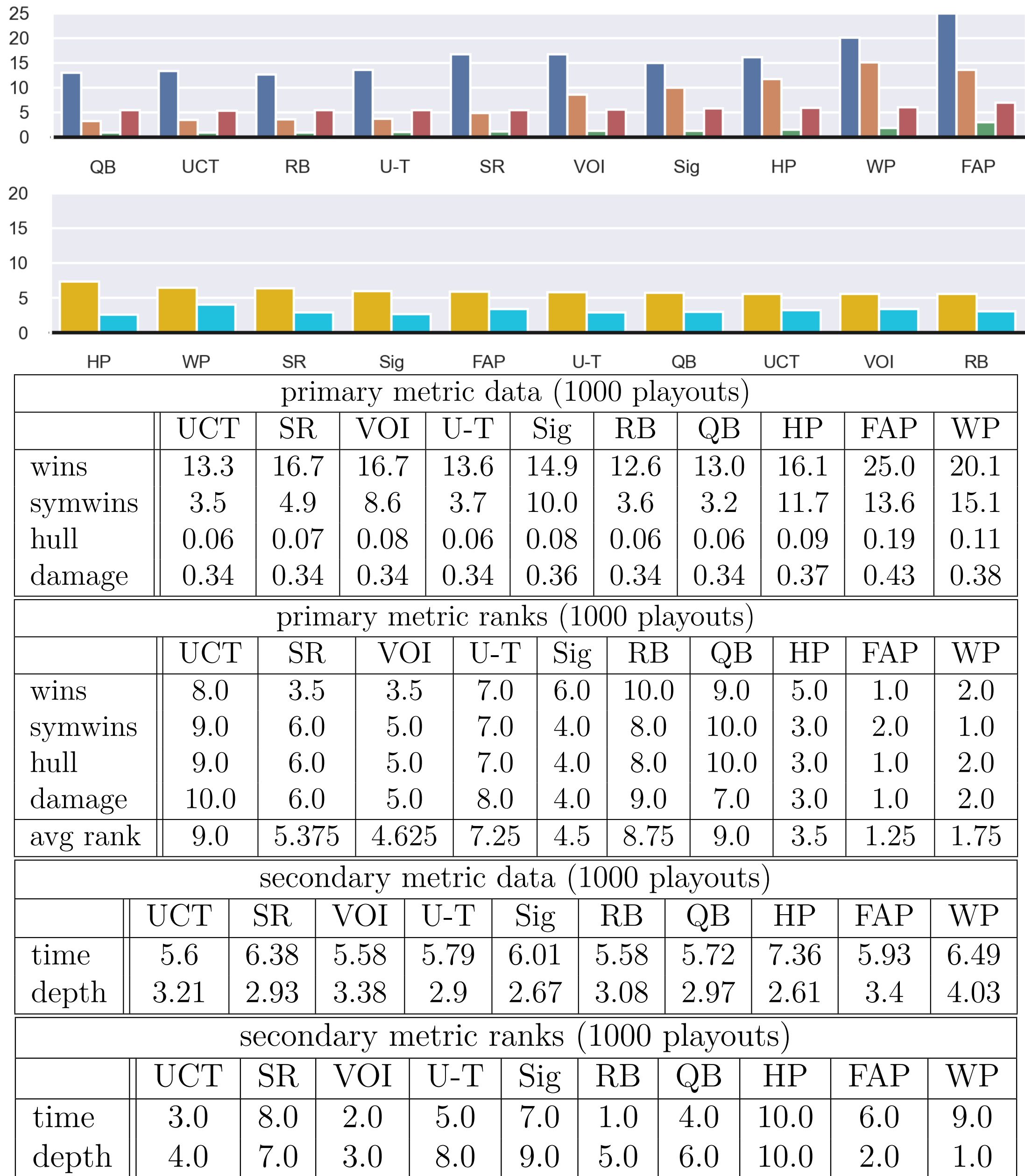
primary metric data (500 playouts)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	14.3	16.1	16.4	14.4	14.3	13.9	14.5	15.7	23.0	19.4
symwins	3.3	5.0	7.8	3.9	9.5	4.0	4.3	12.0	13.7	14.1
hull	0.12	0.14	0.15	0.12	0.15	0.12	0.12	0.17	0.31	0.2
damage	0.69	0.69	0.7	0.7	0.72	0.69	0.7	0.74	0.87	0.75

primary metric ranks (500 playouts)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	8.5	4.0	3.0	7.0	8.5	10.0	6.0	5.0	1.0	2.0
symwins	10.0	6.0	5.0	9.0	4.0	8.0	7.0	3.0	2.0	1.0
hull	9.0	6.0	5.0	7.0	4.0	10.0	8.0	3.0	1.0	2.0
damage	10.0	9.0	5.0	7.0	4.0	8.0	6.0	3.0	1.0	2.0
avg rank	9.375	6.25	4.5	7.5	5.125	9.0	6.75	3.5	1.25	1.75

secondary metric data (500 playouts)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	5.49	5.85	5.77	5.56	5.55	5.31	5.32	6.11	5.83	6.02
depth	3.12	3.1	3.24	2.96	2.79	3.22	2.96	2.69	3.07	4.29

secondary metric ranks (500 playouts)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	3.0	8.0	6.0	5.0	4.0	1.0	2.0	10.0	7.0	9.0
depth	4.0	5.0	2.0	8.0	9.0	3.0	7.0	10.0	6.0	1.0

Figure 8: Data accumulated over tests with the number of playouts set to 500. The first graph shows the average number of wins (blue), symwins (orange), the average remaining hull (orange) and damage dealt to the enemy (red). It is sorted according to the remaining hull. The second graph shows the average iteration time (yellow) and tree depth (cyan). It is sorted according to the former metric. The four tables report specific values of the data and the ranks of the different variants when they are sorted according to a given metric.



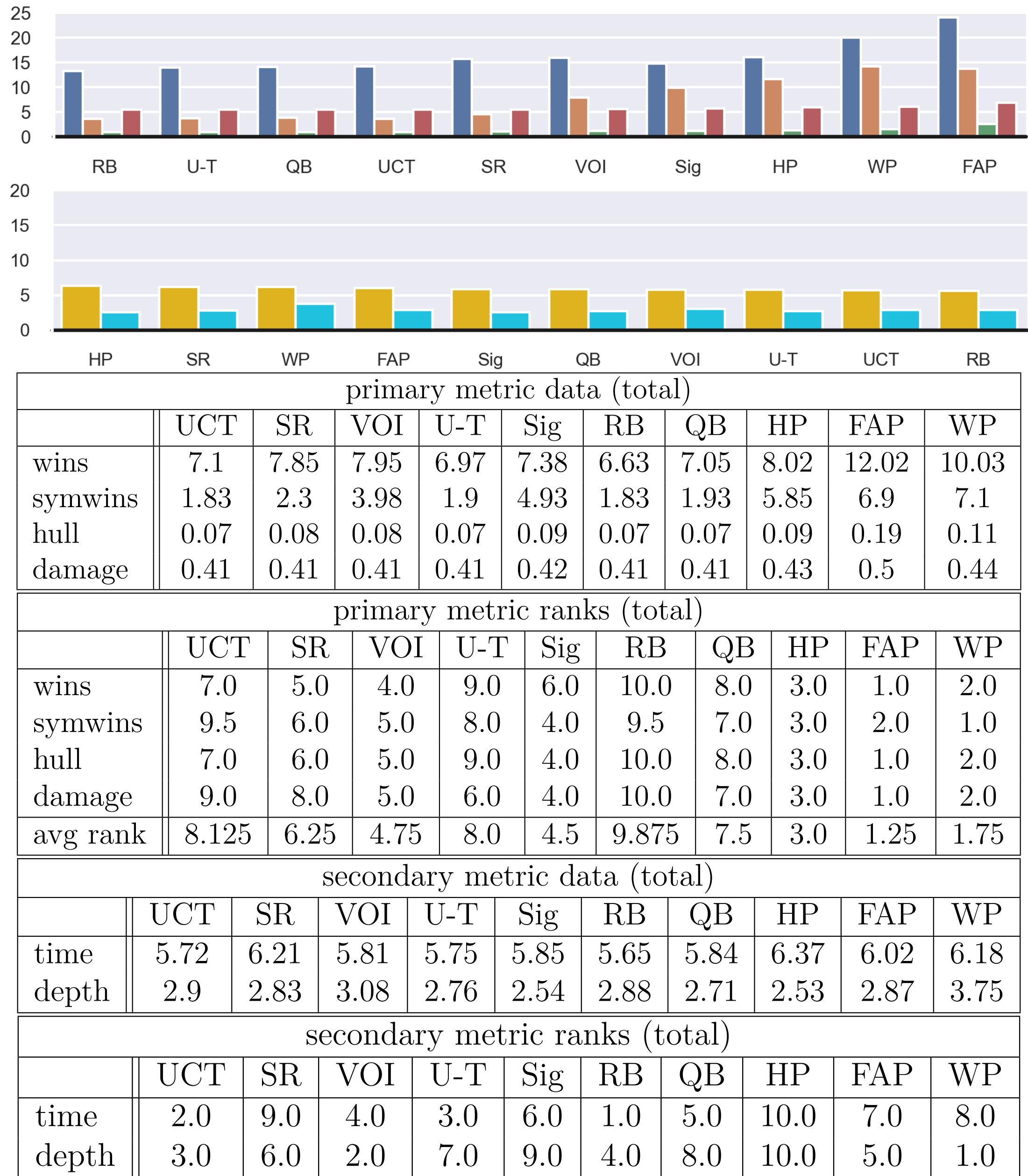
primary metric data (1000 playouts)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	13.3	16.7	16.7	13.6	14.9	12.6	13.0	16.1	25.0	20.1
symwins	3.5	4.9	8.6	3.7	10.0	3.6	3.2	11.7	13.6	15.1
hull	0.06	0.07	0.08	0.06	0.08	0.06	0.06	0.09	0.19	0.11
damage	0.34	0.34	0.34	0.34	0.36	0.34	0.34	0.37	0.43	0.38

primary metric ranks (1000 playouts)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	8.0	3.5	3.5	7.0	6.0	10.0	9.0	5.0	1.0	2.0
symwins	9.0	6.0	5.0	7.0	4.0	8.0	10.0	3.0	2.0	1.0
hull	9.0	6.0	5.0	7.0	4.0	8.0	10.0	3.0	1.0	2.0
damage	10.0	6.0	5.0	8.0	4.0	9.0	7.0	3.0	1.0	2.0
avg rank	9.0	5.375	4.625	7.25	4.5	8.75	9.0	3.5	1.25	1.75

secondary metric data (1000 playouts)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	5.6	6.38	5.58	5.79	6.01	5.58	5.72	7.36	5.93	6.49
depth	3.21	2.93	3.38	2.9	2.67	3.08	2.97	2.61	3.4	4.03

secondary metric ranks (1000 playouts)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	3.0	8.0	2.0	5.0	7.0	1.0	4.0	10.0	6.0	9.0
depth	4.0	7.0	3.0	8.0	9.0	5.0	6.0	10.0	2.0	1.0

Figure 9: Data accumulated over tests with the number of playouts set to 1000. The first graph shows the average number of wins (blue), symwins (orange), the average remaining hull (orange) and damage dealt to the enemy (red). It is sorted according to the remaining hull. The second graph shows the average iteration time (yellow) and tree depth (cyan). It is sorted according to the former metric. The four tables report specific values of the data and the ranks of the different variants when they are sorted according to a given metric.



primary metric data (total)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	7.1	7.85	7.95	6.97	7.38	6.63	7.05	8.02	12.02	10.03
symwins	1.83	2.3	3.98	1.9	4.93	1.83	1.93	5.85	6.9	7.1
hull	0.07	0.08	0.08	0.07	0.09	0.07	0.07	0.09	0.19	0.11
damage	0.41	0.41	0.41	0.41	0.42	0.41	0.41	0.43	0.5	0.44

primary metric ranks (total)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	7.0	5.0	4.0	9.0	6.0	10.0	8.0	3.0	1.0	2.0
symwins	9.5	6.0	5.0	8.0	4.0	9.5	7.0	3.0	2.0	1.0
hull	7.0	6.0	5.0	9.0	4.0	10.0	8.0	3.0	1.0	2.0
damage	9.0	8.0	5.0	6.0	4.0	10.0	7.0	3.0	1.0	2.0
avg rank	8.125	6.25	4.75	8.0	4.5	9.875	7.5	3.0	1.25	1.75

secondary metric data (total)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	5.72	6.21	5.81	5.75	5.85	5.65	5.84	6.37	6.02	6.18
depth	2.9	2.83	3.08	2.76	2.54	2.88	2.71	2.53	2.87	3.75

secondary metric ranks (total)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	2.0	9.0	4.0	3.0	6.0	1.0	5.0	10.0	7.0	8.0
depth	3.0	6.0	2.0	7.0	9.0	4.0	8.0	10.0	5.0	1.0

Figure 10: Data accumulated over all playout settings. The first graph shows the average number of wins (blue), symwins (orange), the average remaining hull (orange) and damage dealt to the enemy (red). It is sorted according to the remaining hull. The second graph shows the average iteration time (yellow) and tree depth (cyan). It is sorted according to the former metric. The four tables report specific values of the data and the ranks of the different variants when they are sorted according to a given metric.

amount. The ranks however are computed from unrounded data, so they can be used to determine which variant actually performed better.

From the four figures presented here, we can see that the performance of the individual variants did not change much based on this metric, with the top five variants remaining constant. We will now go over each of the variants in the order in which we introduced them in section 6.

UCT is the variant that all MCTS adjustments are usually compared to. Given this, it performs surprisingly well at 100 playouts, never dipping below seventh place according to any metric and scoring as high as fifth place on the number of wins. Going by average rank, this variant reaches sixth place in this setting. This may simply show that 100 playouts is too low for the strengths of some other variants to show however, and the results obtained at different playout settings seem to confirm this hypothesis.

At 500 playouts, the performance of UCT drops dramatically, suddenly placing it in the last place according to average rank and at 1000 playouts, it remains there, although it is joined by Qualitative Bonus MCTS.

In total, the numbers this variant reaches are still a bit surprising, but this seems to simply be due to the fact that the results obtained at 100 playouts skew the overall averages in its favour. The overall trend seems to be that UCT is among the worst performing variants, which is to be expected.

As for the average time per iteration and average tree depth, UCT scores surprisingly well, scoring second on the former and third on the latter overall. Its good score with respect to time is probably simply due to the fact that, since all the other variants expand on UCT, they often require some additional computation that slows them down in comparison.

SR+CR MCTS quite consistently occupies the sixth place, except for the 100 playouts setting where it performs worse, underperforming even UCT on all metrics except for average remaining hull. This probably means that at such a low number of playouts, this variant allocates too few playouts to verifying whether the moves it considers good really are. This is supported by the fact that this variant always scores near the bottom with respect to average iteration time - seeing as this variant only differs from normal UCT in how decisions are made at the root and should therefore have almost identical time complexity, this suggests that it mostly makes longer playouts, which means that the playouts start from a place higher up in the search tree.

VOI-aware MCTS mostly oscillates around the fifth place on all metrics. It outperforms SR+CR MCTS which would suggest that out of the two approaches to minimizing simple regret at the root, this is the better one. This is consistent with the findings in the original paper. From the graphs shown in this section, it

can be seen that it has a noticeably higher number of symwins compared to all the variants ranked below it.

Another interesting property of this variant is that it always achieves a high average tree depth which would suggest that it hones in on good moves more quickly than most other variants. Its lower score with respect to average iteration time might be due to the somewhat complicated way the estimates of the value of information provided by playouts is computed.

UCB1-Tuned MCTS does not perform particularly well. Going by average rank, it can be placed at the ninth place, just before UCT. As far as we know, the UCB1-Tuned policy has never actually been tested against basic UCB1, so it may be that the any gain caused by this policy simply is not good enough to outperform the gains provided by the other variants.

As for its average iteration time and tree depth, UCB1-Tuned MCTS is consistently ranked near the bottom on depth but shows rather good results with respect to time. The former hints at significantly different behavior compared to normal UCT which has quite a high score in this metric while the latter points to a similarity between the two - since UCB1-Tuned MCTS does not require much more complex computations than normal UCT, it is comparably fast.

Sigmoid MCTS seems to not be so good with respect to average number of wins, but its other scores place it squarely at the fourth place, right above VOI-aware MCTS. Especially visible from the graphs is again its number of symwins which is markedly higher than in all the variants ranked beneath it.

Its average tree depth is consistently among the lowest which would suggest that it focuses more on exploration than exploitation and thus builds a broader tree. It is probably due to this that its average iteration time also is not remarkable.

Relative Bonus MCTS performs very poorly in all the tested settings and on every metric, making it the worst performing variant of all. This is surprising as in the original paper which tested the variant on six different games, it was reported to give a significant performance boost in five of the six tested games, although each game was tested with different parameters. It may simply be the case that this approach is not well-suited for our setting.

This variant's ranking according to average iteration time is consistently among the best, taking the top spot at both 500 and 1000 playouts while its average depth remain average. It is unclear why this is the case, as its computation complexity should be among the higher ones.

Qualitative Bonus MCTS also does not perform very well, taking the seventh place on average, just above UCB1-Tuned. This variant was reported to show similar performance improvements to Relative Bonus MCTS, so its perfor-

mance is also somewhat surprising, although it does at least outperform UCT.

The average tree depth and iteration time ratings for this variant vary quite a bit but never achieve notable results which is in stark contrast to Relative Bonus MCTS and suggests that the trees these two variants build are quite different.

MCTS_HP occupies a solid third place with only slight deviations. Its mean time and depth results are consistently among the worst which is in contrast with other well-performing variants, namely VOI-aware MCTS and WP MCTS. It therefore seems that both distributing playouts more evenly to build a broader tree and focusing on deeply exploring some moves can lead to good results.

FAP MCTS is the best performing variant by quite a large margin. It clearly outperforms every other variant in every metric, except for symwins where WP MCTS seems to be on par with it. Its mean time and depth ratings are not remarkable however.

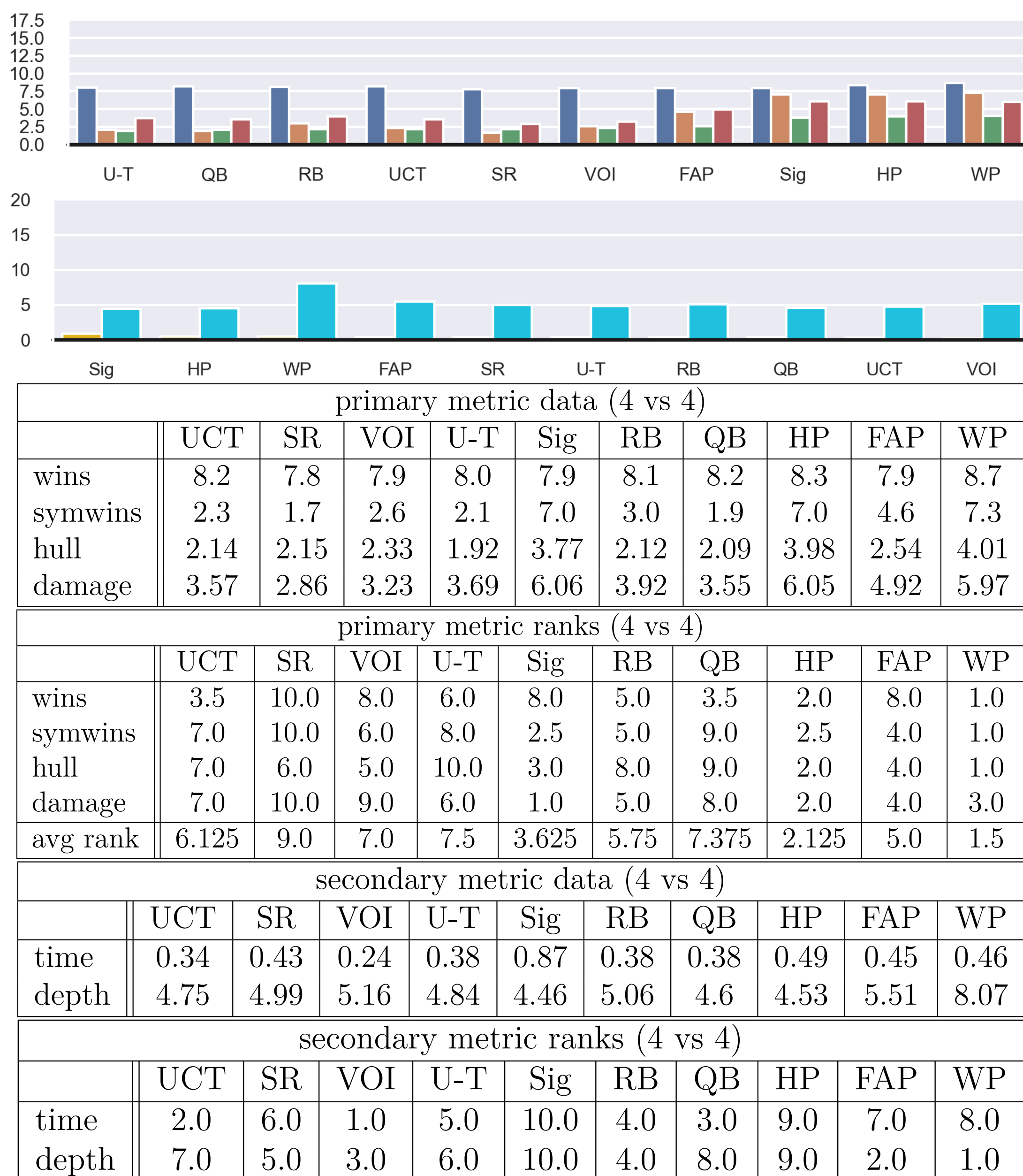
WP MCTS is consistently the second best performing variant. This shows that our assumption about playouts offering more information than just the evaluation of the final state was correct. This variant also permanently holds the top spot with respect to average tree depth while it is always ranked near the bottom with respect to average iteration time. This suggests that even though it builds a deeper tree, the computations it does along the way are too expensive and drag down its average iteration time.

7.3.2 Combat Analysis

In this section, we look at the results separated by battle type. The data reported here are averaged in the same way as in the previous section and the data in the graphs and tables are ordered in the same way too.

At first glance at the graphs, we can see that the results here are much more diverse than in the previous section. We can see some common trends, especially in the graphs that detail the average iteration times and tree depths of the different variants, namely that the former keeps increasing across the different combat settings while the latter keeps decreasing. This is to be expected, as with more units, the state space grows in both breadth and depth which increases the lengths of playouts (and therefore the average iteration time) and makes MCTS focus more on exploration as there is more to explore (which decreases average tree depth).

Another visible trend is the decrease of the remaining hull among all variants with increasing number of units, except for the 64 vs 64 setting where it suddenly jumps up. The decrease is to be expected however, as the number is being averaged over a larger number of units. Imagine, for example, that a 4 vs 4



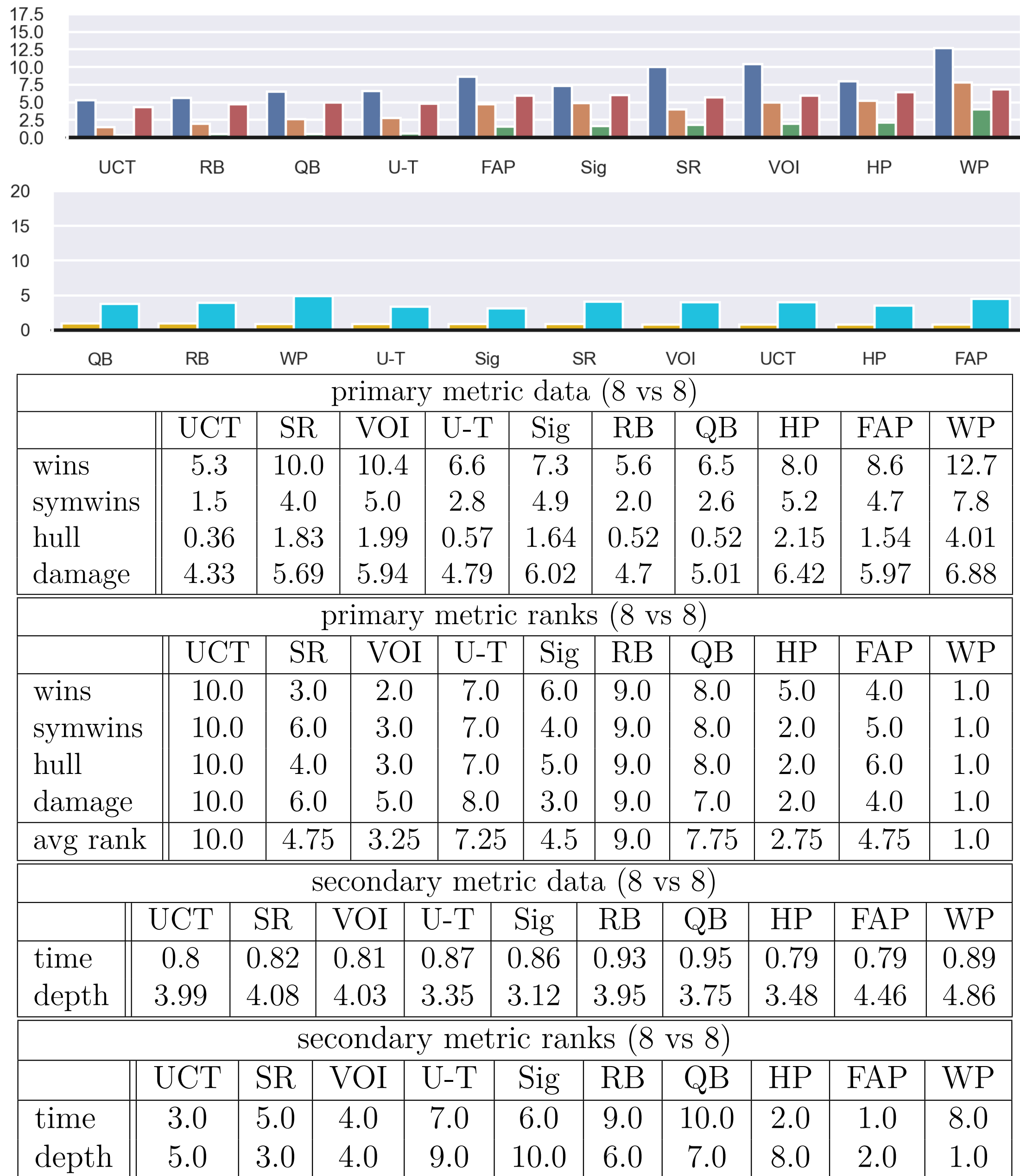
primary metric data (4 vs 4)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	8.2	7.8	7.9	8.0	7.9	8.1	8.2	8.3	7.9	8.7
symwins	2.3	1.7	2.6	2.1	7.0	3.0	1.9	7.0	4.6	7.3
hull	2.14	2.15	2.33	1.92	3.77	2.12	2.09	3.98	2.54	4.01
damage	3.57	2.86	3.23	3.69	6.06	3.92	3.55	6.05	4.92	5.97

primary metric ranks (4 vs 4)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	3.5	10.0	8.0	6.0	8.0	5.0	3.5	2.0	8.0	1.0
symwins	7.0	10.0	6.0	8.0	2.5	5.0	9.0	2.5	4.0	1.0
hull	7.0	6.0	5.0	10.0	3.0	8.0	9.0	2.0	4.0	1.0
damage	7.0	10.0	9.0	6.0	1.0	5.0	8.0	2.0	4.0	3.0
avg rank	6.125	9.0	7.0	7.5	3.625	5.75	7.375	2.125	5.0	1.5

secondary metric data (4 vs 4)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	0.34	0.43	0.24	0.38	0.87	0.38	0.38	0.49	0.45	0.46
depth	4.75	4.99	5.16	4.84	4.46	5.06	4.6	4.53	5.51	8.07

secondary metric ranks (4 vs 4)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	2.0	6.0	1.0	5.0	10.0	4.0	3.0	9.0	7.0	8.0
depth	7.0	5.0	3.0	6.0	10.0	4.0	8.0	9.0	2.0	1.0

Figure 11: Data measured in the 4 vs 4 combat setting. The first graph shows the average numbers of wins (blue), symwins (orange), the average remaining hull (orange) and damage dealt to the enemy (red). It is sorted according to remaining hull. The second graph shows the average iteration time (yellow) and tree depth (cyan). It is sorted according to the former metric. The four tables report specific values of the data and the ranks of the different variants when they are sorted according to a given metric.



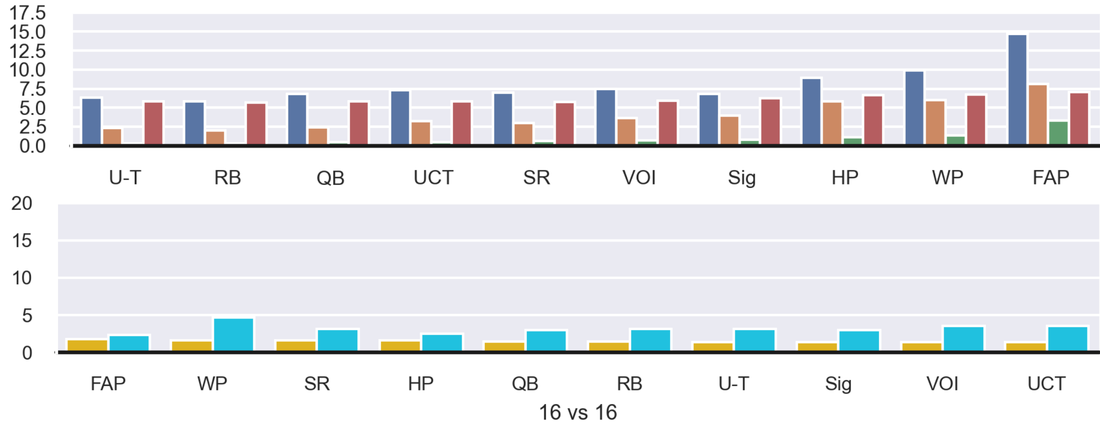
primary metric data (8 vs 8)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	5.3	10.0	10.4	6.6	7.3	5.6	6.5	8.0	8.6	12.7
symwins	1.5	4.0	5.0	2.8	4.9	2.0	2.6	5.2	4.7	7.8
hull	0.36	1.83	1.99	0.57	1.64	0.52	0.52	2.15	1.54	4.01
damage	4.33	5.69	5.94	4.79	6.02	4.7	5.01	6.42	5.97	6.88

primary metric ranks (8 vs 8)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	10.0	3.0	2.0	7.0	6.0	9.0	8.0	5.0	4.0	1.0
symwins	10.0	6.0	3.0	7.0	4.0	9.0	8.0	2.0	5.0	1.0
hull	10.0	4.0	3.0	7.0	5.0	9.0	8.0	2.0	6.0	1.0
damage	10.0	6.0	5.0	8.0	3.0	9.0	7.0	2.0	4.0	1.0
avg rank	10.0	4.75	3.25	7.25	4.5	9.0	7.75	2.75	4.75	1.0

secondary metric data (8 vs 8)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	0.8	0.82	0.81	0.87	0.86	0.93	0.95	0.79	0.79	0.89
depth	3.99	4.08	4.03	3.35	3.12	3.95	3.75	3.48	4.46	4.86

secondary metric ranks (8 vs 8)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	3.0	5.0	4.0	7.0	6.0	9.0	10.0	2.0	1.0	8.0
depth	5.0	3.0	4.0	9.0	10.0	6.0	7.0	8.0	2.0	1.0

Figure 12: Data measured in the 8 vs 8 combat setting. The first graph shows the average numbers of wins (blue), symwins (orange), the average remaining hull (orange) and damage dealt to the enemy (red). It is sorted according to remaining hull. The second graph shows the average iteration time (yellow) and tree depth (cyan). It is sorted according to the former metric. The four tables report specific values of the data and the ranks of the different variants when they are sorted according to a given metric.



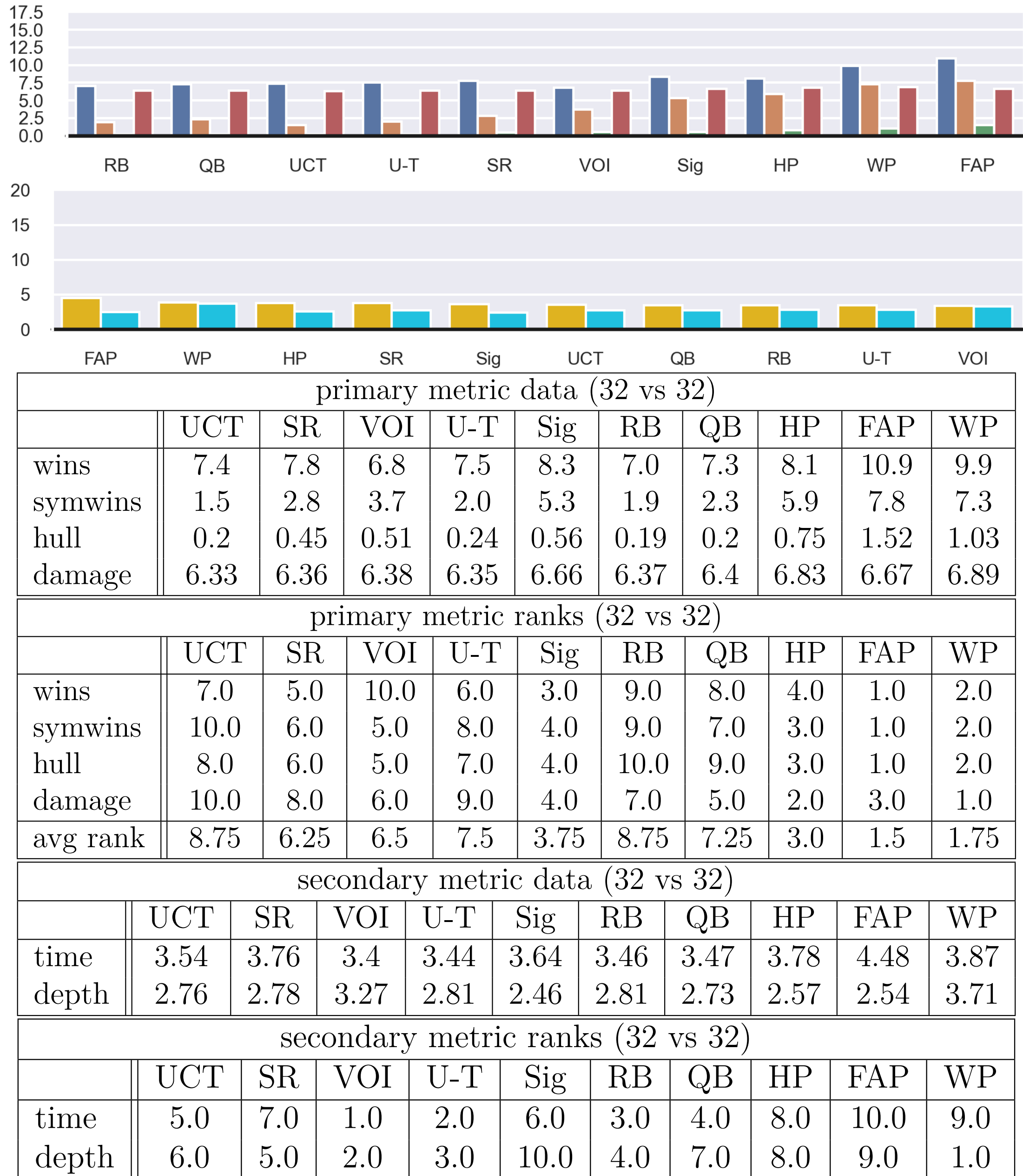
primary metric data (16 vs 16)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	7.3	7.0	7.5	6.3	6.8	5.8	6.8	8.9	14.7	9.9
symwins	3.2	3.0	3.6	2.3	4.0	2.0	2.4	5.8	8.1	6.0
hull	0.47	0.63	0.69	0.29	0.77	0.34	0.43	1.1	3.32	1.4
damage	5.86	5.72	5.9	5.84	6.2	5.71	5.8	6.65	7.02	6.72

primary metric ranks (16 vs 16)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	5.0	6.0	4.0	9.0	7.5	10.0	7.5	3.0	1.0	2.0
symwins	6.0	7.0	5.0	9.0	4.0	10.0	8.0	3.0	1.0	2.0
hull	7.0	6.0	5.0	10.0	4.0	9.0	8.0	3.0	1.0	2.0
damage	6.0	9.0	5.0	7.0	4.0	10.0	8.0	3.0	1.0	2.0
avg rank	6.0	7.0	4.75	8.75	4.875	9.75	7.875	3.0	1.0	2.0

secondary metric data (16 vs 16)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	1.37	1.61	1.39	1.41	1.41	1.47	1.5	1.61	1.82	1.66
depth	3.56	3.2	3.59	3.18	2.98	3.17	3.04	2.52	2.37	4.67

secondary metric ranks (16 vs 16)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	1.0	8.0	2.0	4.0	3.0	5.0	6.0	7.0	10.0	9.0
depth	3.0	4.0	2.0	5.0	8.0	6.0	7.0	9.0	10.0	1.0

Figure 13: Data measured in the 16 vs 16 combat setting. The first graph shows the average numbers of wins (blue), symwins (orange), the average remaining hull (orange) and damage dealt to the enemy (red). It is sorted according to remaining hull. The second graph shows the average iteration time (yellow) and tree depth (cyan). It is sorted according to the former metric. The four tables report specific values of the data and the ranks of the different variants when they are sorted according to a given metric.



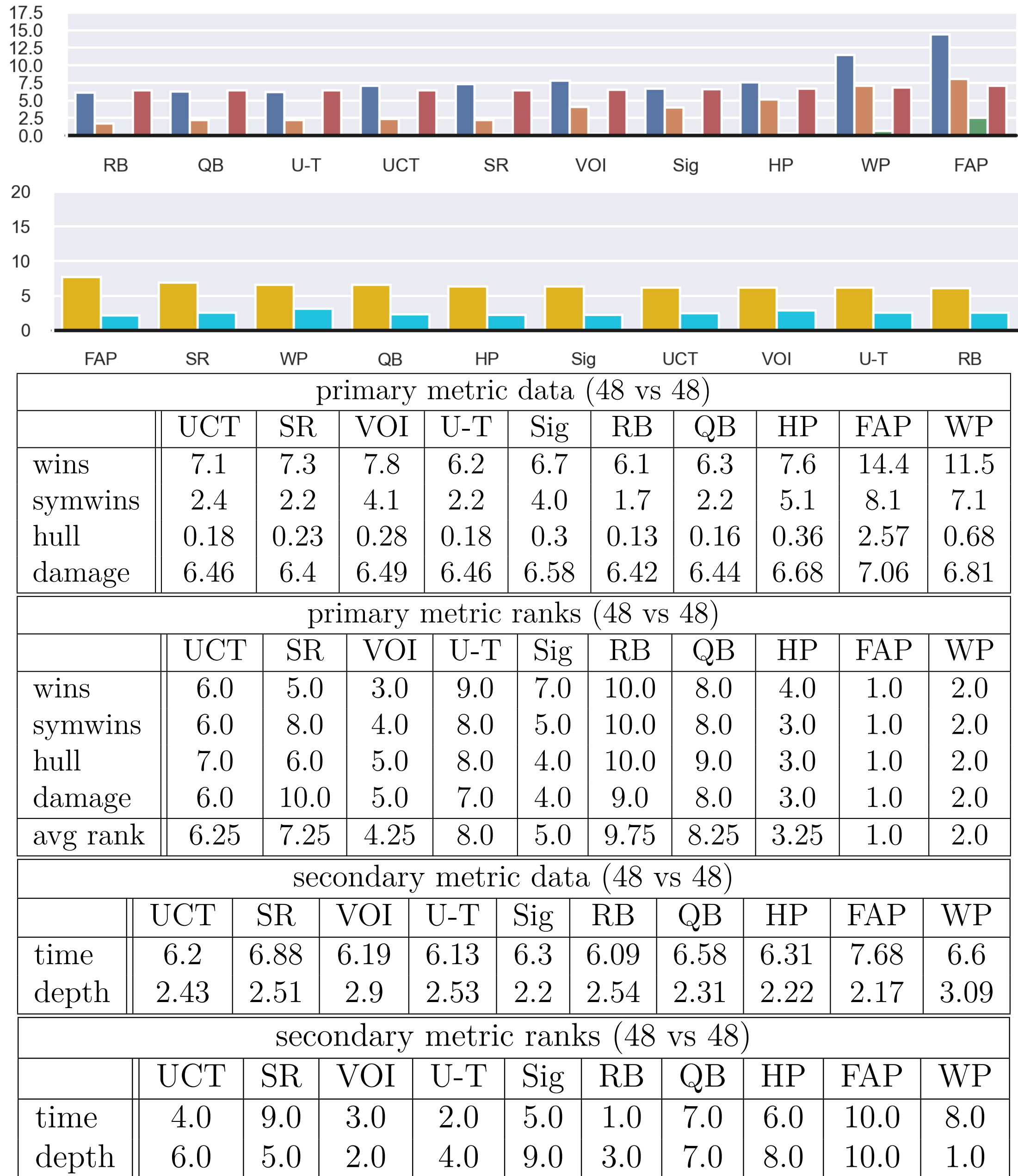
primary metric data (32 vs 32)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	7.4	7.8	6.8	7.5	8.3	7.0	7.3	8.1	10.9	9.9
symwins	1.5	2.8	3.7	2.0	5.3	1.9	2.3	5.9	7.8	7.3
hull	0.2	0.45	0.51	0.24	0.56	0.19	0.2	0.75	1.52	1.03
damage	6.33	6.36	6.38	6.35	6.66	6.37	6.4	6.83	6.67	6.89

primary metric ranks (32 vs 32)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	7.0	5.0	10.0	6.0	3.0	9.0	8.0	4.0	1.0	2.0
symwins	10.0	6.0	5.0	8.0	4.0	9.0	7.0	3.0	1.0	2.0
hull	8.0	6.0	5.0	7.0	4.0	10.0	9.0	3.0	1.0	2.0
damage	10.0	8.0	6.0	9.0	4.0	7.0	5.0	2.0	3.0	1.0
avg rank	8.75	6.25	6.5	7.5	3.75	8.75	7.25	3.0	1.5	1.75

secondary metric data (32 vs 32)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	3.54	3.76	3.4	3.44	3.64	3.46	3.47	3.78	4.48	3.87
depth	2.76	2.78	3.27	2.81	2.46	2.81	2.73	2.57	2.54	3.71

secondary metric ranks (32 vs 32)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	5.0	7.0	1.0	2.0	6.0	3.0	4.0	8.0	10.0	9.0
depth	6.0	5.0	2.0	3.0	10.0	4.0	7.0	8.0	9.0	1.0

Figure 14: Data measured in 32 vs 32 combat setting. The first graph shows the average numbers of wins (blue), symwins (orange), the average remaining hull (orange) and damage dealt to the enemy (red). It is sorted according to remaining hull. The second graph shows the average iteration time (yellow) and tree depth (cyan). It is sorted according to the former metric. The four tables report specific values of the data and the ranks of the different variants when they are sorted according to a given metric.



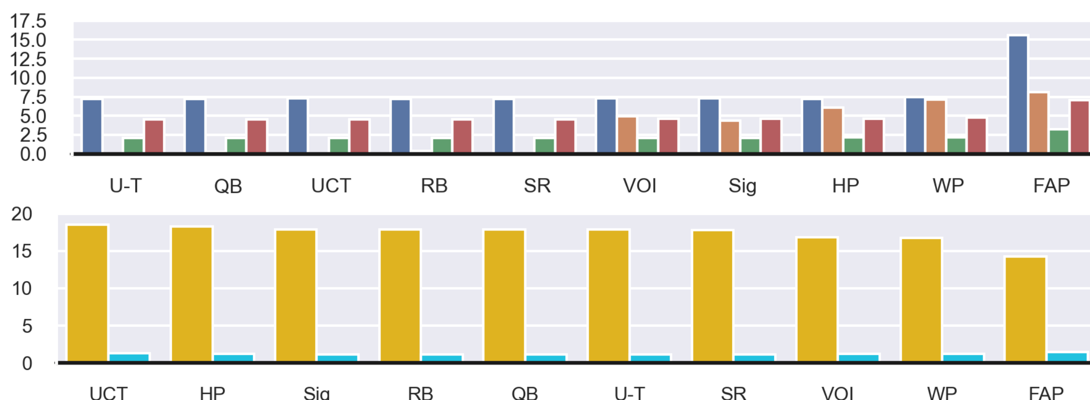
primary metric data (48 vs 48)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	7.1	7.3	7.8	6.2	6.7	6.1	6.3	7.6	14.4	11.5
symwins	2.4	2.2	4.1	2.2	4.0	1.7	2.2	5.1	8.1	7.1
hull	0.18	0.23	0.28	0.18	0.3	0.13	0.16	0.36	2.57	0.68
damage	6.46	6.4	6.49	6.46	6.58	6.42	6.44	6.68	7.06	6.81

primary metric ranks (48 vs 48)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	6.0	5.0	3.0	9.0	7.0	10.0	8.0	4.0	1.0	2.0
symwins	6.0	8.0	4.0	8.0	5.0	10.0	8.0	3.0	1.0	2.0
hull	7.0	6.0	5.0	8.0	4.0	10.0	9.0	3.0	1.0	2.0
damage	6.0	10.0	5.0	7.0	4.0	9.0	8.0	3.0	1.0	2.0
avg rank	6.25	7.25	4.25	8.0	5.0	9.75	8.25	3.25	1.0	2.0

secondary metric data (48 vs 48)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	6.2	6.88	6.19	6.13	6.3	6.09	6.58	6.31	7.68	6.6
depth	2.43	2.51	2.9	2.53	2.2	2.54	2.31	2.22	2.17	3.09

secondary metric ranks (48 vs 48)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	4.0	9.0	3.0	2.0	5.0	1.0	7.0	6.0	10.0	8.0
depth	6.0	5.0	2.0	4.0	9.0	3.0	7.0	8.0	10.0	1.0

Figure 15: Data measured over in the 48 vs 48 combat setting. The first graph shows the average numbers of wins (blue), symwins (orange), the average remaining hull (orange) and damage dealt to the enemy (red). It is sorted according to remaining hull. The second graph shows the average iteration time (yellow) and tree depth (cyan). It is sorted according to the former metric. The four tables report specific values of the data and the ranks of the different variants when they are sorted according to a given metric.



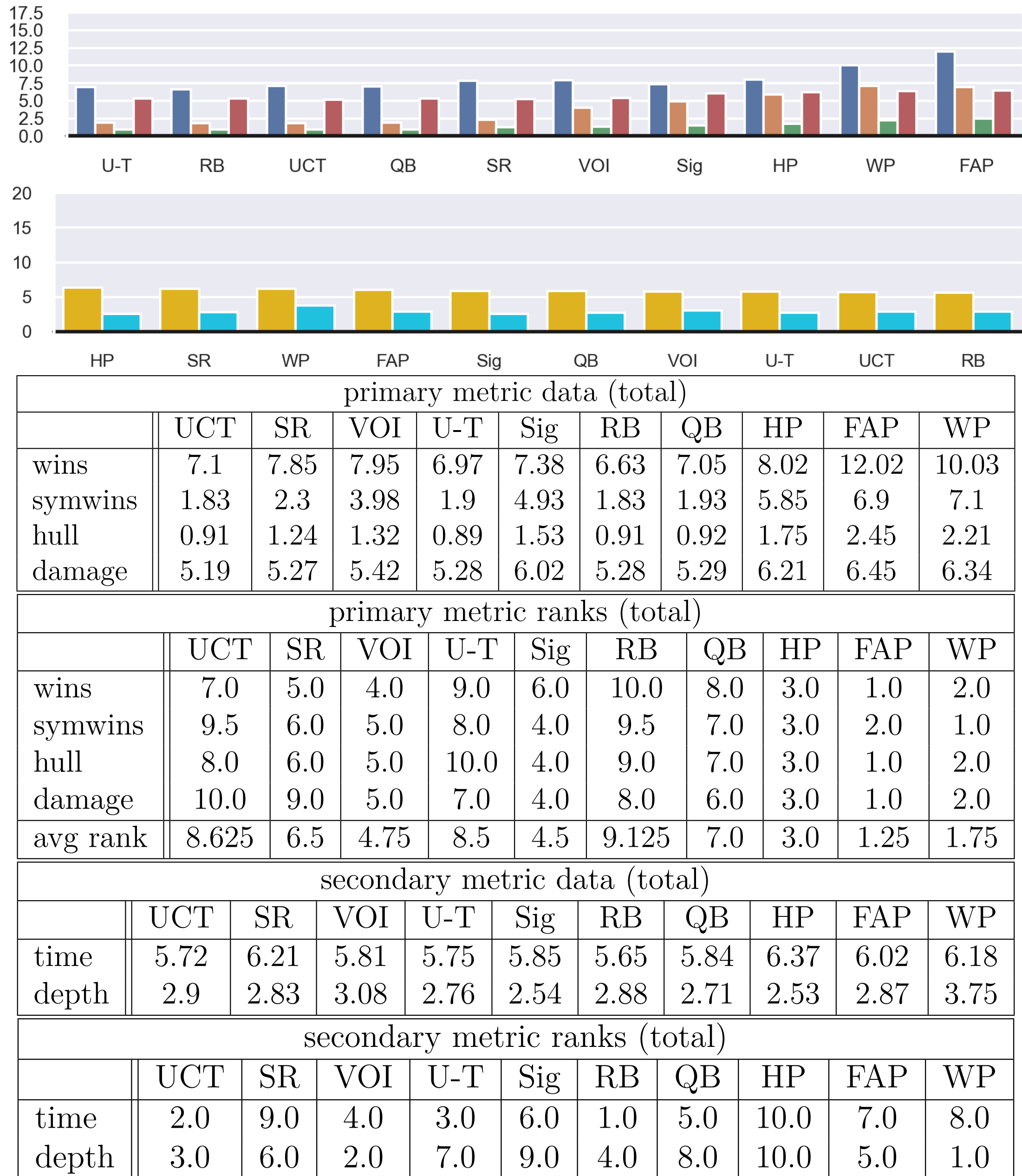
primary metric data (64 vs 64)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	7.3	7.2	7.3	7.2	7.3	7.2	7.2	7.2	15.6	7.5
symwins	0.1	0.1	4.9	0.0	4.4	0.4	0.2	6.1	8.1	7.1
hull	2.13	2.13	2.13	2.13	2.13	2.13	2.13	2.14	3.24	2.15
damage	4.56	4.56	4.58	4.56	4.58	4.56	4.56	4.65	7.08	4.74

primary metric ranks (64 vs 64)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	4.0	8.0	4.0	8.0	4.0	8.0	8.0	8.0	1.0	2.0
symwins	8.5	8.5	4.0	10.0	5.0	6.0	7.0	3.0	1.0	2.0
hull	8.0	6.0	5.0	10.0	4.0	7.0	9.0	3.0	1.0	2.0
damage	6.0	9.0	4.0	7.0	5.0	8.0	10.0	3.0	1.0	2.0
avg rank	6.625	7.875	4.25	8.75	4.5	7.25	8.5	4.25	1.0	2.0

secondary metric data (64 vs 64)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	18.49	17.79	16.8	17.85	17.88	17.88	17.87	18.3	14.28	16.73
depth	1.32	1.15	1.22	1.16	1.17	1.16	1.16	1.27	1.45	1.25

secondary metric ranks (64 vs 64)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	10.0	4.0	3.0	5.0	8.0	7.0	6.0	9.0	1.0	2.0
depth	2.0	10.0	5.0	8.0	6.0	8.0	8.0	3.0	1.0	4.0

Figure 16: Data measured in the 64 vs 64 combat setting. The first graph shows the average numbers of wins (blue), symwins (orange), the average remaining hull (orange) and damage dealt to the enemy (red). It is sorted according to remaining hull. The second graph shows the average iteration time (yellow) and tree depth (cyan). It is sorted according to the former metric. The four tables report specific values of the data and the ranks of the different variants when they are sorted according to a given metric.



primary metric data (total)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	7.1	7.85	7.95	6.97	7.38	6.63	7.05	8.02	12.02	10.03
symwins	1.83	2.3	3.98	1.9	4.93	1.83	1.93	5.85	6.9	7.1
hull	0.91	1.24	1.32	0.89	1.53	0.91	0.92	1.75	2.45	2.21
damage	5.19	5.27	5.42	5.28	6.02	5.28	5.29	6.21	6.45	6.34

primary metric ranks (total)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
wins	7.0	5.0	4.0	9.0	6.0	10.0	8.0	3.0	1.0	2.0
symwins	9.5	6.0	5.0	8.0	4.0	9.5	7.0	3.0	2.0	1.0
hull	8.0	6.0	5.0	10.0	4.0	9.0	7.0	3.0	1.0	2.0
damage	10.0	9.0	5.0	7.0	4.0	8.0	6.0	3.0	1.0	2.0
avg rank	8.625	6.5	4.75	8.5	4.5	9.125	7.0	3.0	1.25	1.75

secondary metric data (total)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	5.72	6.21	5.81	5.75	5.85	5.65	5.84	6.37	6.02	6.18
depth	2.9	2.83	3.08	2.76	2.54	2.88	2.71	2.53	2.87	3.75

secondary metric ranks (total)										
	UCT	SR	VOI	U-T	Sig	RB	QB	HP	FAP	WP
time	2.0	9.0	4.0	3.0	6.0	1.0	5.0	10.0	7.0	8.0
depth	3.0	6.0	2.0	7.0	9.0	4.0	8.0	10.0	5.0	1.0

Figure 17: Data measured over all the combat settings. The first graph shows the average numbers of wins (blue), symwins (orange), the average remaining hull (orange) and damage dealt to the enemy (red). It is sorted according to remaining hull. The second graph shows the average iteration time (yellow) and tree depth (cyan). It is sorted according to the former metric. The four tables report specific values of the data and the ranks of the different variants when they are sorted according to a given metric.

battle ends with the winner having just one unit with full hull remaining. This could mean that the two sides were pretty evenly matched. If a 64 vs 64 battle ends with $64/4 = 16$ units remaining on one side, then that side has achieved a pretty substantial victory. This is why it is quite surprising that the remaining hull suddenly jumps up in the 64 vs 64 combat setting. It is possible that the randomly generated positions we used somehow allow a variant which gets into a slightly advantageous position to win quickly or that the state space suddenly becomes too large for most variants to explore properly and the battles become more random which leads to stronger wins for agents that happen to stumble upon good moves.

We can also observe that the average number of wins seems to diversify starting at the 8 vs 8 and then approach the same mean in all the variants except for FAP MCTS. This seems to support our previous hypothesis that most variants are not good at dealing with large state spaces which also causes them to approach the same performance as the state space gets larger.

We once again go over all the variants in the order in which they were introduced and analyse what the results mean for them.

UCT's performance is quite inconsistent throughout the different settings. In battles of 4 against 4, it achieves a surprisingly high average number of wins, but performs poorly on all other metrics. The average number of wins in this setting has a pretty narrow range of values however (from 7.8 to 8.7, although the upper bound actually goes down to 8.3 if we discard the best performing variant), so UCT's good score may just be due to variance of the results.

In the 8 vs 8 setting, UCT suddenly drops down to last place on every metric. When we move on to the 16 vs 16 setting however, we see that UCT jumped back up to around the sixth place and keeps oscillating around that place in the remaining settings as well. It may therefore be the case that the positions we generated in the 8 vs 8 setting were somehow less advantageous to UCT or more advantageous to some other variants, or that the size of the state space was ideal for some variants.

As we mentioned in the previous section, the fact that UCT performs this well quite consistently is pretty surprising, given that all the other variants are supposed to present improvements over it.

UCT's rankings with respect to mean iteration time and tree depth vary wildly, although this may mostly be due to variance of the results as the range of values of these metrics is quite narrow.

SR+CR MCTS starts out with a very poor performance in the 4 vs 4 setting, but them jumps up to around the fifth place at 8 vs 8 and descends to around the sixth place where it remains until the 64 vs 64 setting where its

performance drops again. The one place drop from 8 vs 8 to 16 vs 16 can probably be explained by the fact that FAP MCTS improves drastically between these two settings, moving to first place, so it shifts all the variants that were before it by one place (although SR+CR MCTS outperformed it in only 2 out of four metrics in the 8 vs 8 setting).

It is unclear why this variant performs so poorly in the first setting. Perhaps, since the state space is quite small, the other variants simply have a chance to explore it better and therefore this variant's strengths do not come into play.

Its time and depth rankings seem to mostly be quite average across the different settings.

VOI-aware MCTS, just like SR+CR MCTS, does not perform very well in the 4 vs 4 combat setting and then improves greatly in the 8 vs 8 setting, going up to around third place on average and then drops one place, possibly due to the improvement of FAP MCTS. This behavior is similar to SR+CR MCTS which suggests that it is caused by what they have in common - attempting to minimize simple regret at the root.

A notable property of this variant is that is keeps a relatively high number of symwins even in the 64 vs 64 setting where all the variants ranked beneath it with respect to this metric except for Sigmoid MCTS have almost none which means that its wins are usually better than those of its opponents.

As for its mean time and depth performance, it is among the higher ranked variants in most settings, especially in the former category and never reaches a lower place than fifth on either metric. Like we mentioned in the previous section, this seems to imply that it hones in faster on moves it considers to be good and explores them in more depth.

UCB1-Tuned MCTS performs quite poorly in all settings, usually underperforming UCT. This is consistent with our observations from the previous section. Its time and depth rankings also do not seem particularly remarkable.

Sigmoid MCTS seems to perform quite well in all metrics, usually placing at around the fourth place, except for the average number of wins, where it usually does not achieve such good results.

Its depth rankings are consistently among the worst, except for the 64 vs 64 setting, while its time rankings mostly oscillate around the middle. The latter is probably due to the fact that shallow trees mean longer playouts.

Relative Bonus MCTS achieves some notable results in the 4 vs 4 settings, but then drops on all measured metrics and stays near the bottom in all settings except for 64 vs 64 where there are three variants with an average ranking worse than it, although, as we explained previously, this is probably just due to some of the different variants approaching the same level of performance as the state

space grows larger.

This variant has very inconsistent rankings with respect to average iteration time and average tree depth. It is unclear why this is the case.

Qualitative Bonus MCTS shows no significant achievements in any setting, always oscillating around the eighth place on almost all metrics and its time and depth rankings are also unremarkable.

MCTS_HP consistently achieves good results, always placing itself right behind WP MCTS. It is notable that it consistently achieves good scores on all primary metrics except for the number of wins. Its time and depth rankings are usually near the bottom which is consistent with the data from the previous section.

FAP MCTS starts out weak, achieving no particularly notable results in either the 4 vs 4 or the 8 vs 8 settings. However, starting with 16 vs 16, it takes the number one spot and stays there, clearly becoming the best performing variant on average. It is also the only variant which seems to be good at dealing with large state spaces as is evident from its performance in the 64 vs 64 setting where it most notably achieves an average number of wins that is more than double the second best variant. It is also interesting to note however that its number of symwins does not show the same boost and remains comparable to WP MCTS and MCTS_HP. This means that in a situation when it loses once and wins once, its hull is most likely lower than that of its opponent, meaning that it probably does not achieve very strong victories.

The average iteration time of this variant is usually ranked low, with the exception of the 8 vs 8 and 64 vs 64 setting. Its average depth rating shows similar behavior, except it achieves second place in the 4 vs 4 setting. It is unclear why it achieves such good results according to these metrics in the 8 vs 8 setting, but its good results in the 64 vs 64 setting are just further testament to the fact that it is the only variant capable o dealing with a state space of such size.

Meanwhile its poor performance in the other settings may be due to the fact that, since it treats a single playout as many playouts, depending on when it occurred, the path from the root to the node where the playout was started may be seen by the algorithm as sufficiently explored after a single playout, therefore encouraging exploration a lot more.

WP MCTS is the only variant that consistently performs well on all primary metrics. In overall performance, it is second only to FAP MCTS, although it also seems to be incapable of effectively dealing with large state space and its performance devolves towards a common mean with increasing state space size. Most notable is its high number of symwins which, in the 64 vs 64 setting, almost

matches its number of wins, meaning that even in situations when it loses once and wins once, it usually achieves a stronger win than its opponent.

This variant also consistently holds the first place in the average depth rankings, except for the 64 vs 64 setting, which, in combination with its high performance, suggests that it identifies good moves more quickly than most other variants.

Its average iteration time, on the other hand, is always ranked near the bottom (except for the 64 vs 64 setting), which is probably due to the fact that the computation it has to do at the end of every playout is rather expensive.

8 Conclusions and Future Work

8.1 Conclusions

In this thesis, we have gone over ten variants of the Monte Carlo Tree Search algorithm, including one of our own making, and compared them in the context of a 4X computer game called Children of the Galaxy. Since the branching factor of this game was too large, we resorted to doing our search in script space.

We performed a series of preliminary tests to select the best algorithm settings for all the variants and picked one setting for each one. We then tested all the variants with their chosen algorithm settings against each other in various combat settings and with various numbers of playouts at their disposal. During these tests, we measured their average number of wins, symwins, their remaining hull, the damage they dealt to the enemy, their average iteration time and their average tree depth. The results showed a pretty consistent ranking of the first six variants. These variants were FAP MCTS, WP MCTS, MCTS_HP, Sigmoid MCTS, VOI MCTS and SR+CR MCTS in that order. The remaining variants - Relative Bonus MCTS, Qualitative Bonus MCTS, UCT and UCB1-Tuned MCTS did not show particularly good results in any setting that we tested and almost always occupied the bottom four spots according to most metrics that we measured, although their ordering was not as clearly determined.

The fact that UCT was among the worst performing variants that we tried is a good thing, since it represents the default implementation of MCTS that all the others were supposed to present improvements over. It is therefore a bit surprising that it did not consistently occupy the lowest rank. We ran enough tests to rule out the possibility of this being just due to variance of the data with a large degree of certainty. It might be the case however, that Relative Bonus MCTS, Qualitative Bonus MCTS and UCB1-Tuned MCTS simply were not suited for our chosen environment and would perform better in some other environment.

Our own variant, WP MCTS, which we introduced in the paper, was based on the idea that a playout can offer more information than just the final state and that the entire trajectory should be taken into account instead. We found that this variant had the most consistently good results of all the tested variants, occupying the first or second spot according to every primary metric independent of how we separated the data with only one exception - it was ranked third according to damage dealt to the enemy in the 4 vs 4 combat setting.

8.2 Future Work

Obvious possibilities for future work are comparing the variants we tested here in different settings to see whether the results reported here still hold and comparing them against algorithms that are not based on MCTS, like Portfolio Greedy Search [4].

Another area for possible research is trying out different combinations of these variants. As the variants tested by us often augment different parts of the original algorithm, it should be possible to combine multiple of them into one algorithm and test if doing so yields improved results.

As for future work regarding WP MCTS, we only tested a few approaches to how weights can be assigned to states encountered during a playout, so this is definitely another topic that can be explored further. It would be especially interesting to test whether best results are achieved if all states are treated as equally ”important”, or if there are some states which should be considered to contain more valuable information than others.

Bibliography

[1] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multi-armed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[2] G. V. D. Broeck, K. Driessens, and J. Ramon. Monte-carlo tree search in poker using expected reward distributions. In *Asian Conference on Machine Learning*, pages 367–381, 2009.

[3] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[4] D. Churchill and M. Buro. Portfolio greedy search and simulation for large-scale combat in starcraft. In *Proceedings of the Conference on Computational Intelligence in Games*, pages 1–8, 2013.

[5] R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, volume 26, pages 72–83, 2006.

[6] Dennis Dacoste. The future of chess-playing technologies and the significance of kasparov versus deep blue. *The Computer Journal*, 1997.

[7] S. C. Huang, B. Arneson, R. B. Hayward, M. Müller, and J. Pawlewicz. Mohex 2.0: a pattern-based mcts hex player. In *International Conference on Computers and Games*, pages 60–71, 2013.

[8] N. Justesen, T. Mahlmann, and J. Togelius. Online evolution for multi-action adversarial games. In *European Conference on the Applications of Evolutionary Computation*, pages 590–603, 2016.

[9] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293, 2006.

[10] T. Kozelek. Methods of mcts and the game arimaa. Master's thesis, Charles University, 2009.

[11] S. Ontañón. Informed monte carlo tree search for real-time strategy games. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2016.

[12] S. Ontañón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss. A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in games*, 5(4):293–311, 2013.

[13] T. Pepels, M. J. Tak, M. Lanctot, and M. H. Winands. Quality-based rewards for monte-carlo tree search simulations. In *ECAI*, pages 705–710, 2014.

[14] T. Pepels, M. H. Winands, and M. Lanctot. Real-time monte carlo tree search in ms pac-man. *Transactions on Computational Intelligence and AI in games*, 6(3):245–257, 2014.

[15] K. Shibahara and Y. Kotani. Combining final score with winning percentage by sigmoid function in monte-carlo simulations. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 183–190, 2008.

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

[17] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[18] D. Tolpin and S. Shimony. Mcts based on simple regret. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, pages 570–576, 2012.

[19] F. Xie and Z. Liu. Mcts based on simple regret. In *2009 Third International Symposium on Intelligent Information Technology Application*, volume 2, pages 125–128, 2009.

[20] M. Świechowski, K. Godlewski, B. Sawicki, and J. Mańdziuk. Monte carlo tree search: A review of recent modifications and applications. *arXiv preprint*, 2021.

[21] P. Šmejkal and 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.

A User Documentation

Since the original project was created by Pavel Šmejkal, we omit its user documentation and instead refer the reader to the one created by its author[2].

We left the original code mostly intact with only a few changes. Most notably, we added scripts for representing the different Monte Carlo Tree Search that we were testing (with the exceptions of UCT and MCTS_HP, as they were already present) and made a few small adjustments to the original MCTS script. The changes made here were for one of two reasons - to make it possible for the new variants to override inherited methods and to change what data was being outputted.

Since we were adding new variants which were supposed to be configurable using xml files, we also had to expand the BenchmarkFactory class, which is responsible for parsing xml configurations, so that it would properly parse the configurations for our variants.

We also tinkered with the ExecuteMicro.cs script when debugging different variants, as this script calls the variants during every round of gameplay, and adjusted the Game and GameResult classes to allow for the accumulation of intermediate values during playouts.

Besides the main project however, we have created a number of programs to help with processing data and include them in case they might be useful to anyone who would try to follow up on our work. Please keep in mind that many of these programs were created hastily with the purpose of quickly processing some data and are far from polished. Except for the XML Manager program, none of them have UI, so any changes in behaviour must be done by adjusting the source code.

Before we can explain the workings of these programs however, we need to go over the data that we worked with.

A.1 Data and Files

Our data were gathered in two types of files - .txt files that contained measurements of an algorithm's iteration time and tree depth and .csv tables which contained various metrics, including the numbers of wins, symwins and the amount of remaining hull, all separated by combat setting. For every test that we ran, one of each of these files was created with the former being stored in a folder called {*algorithm name*}/*data*, the path to which is specifiable in the Mcts.cs file and the latter being stored in a folder called *results*, present in the same directory as the executable.

[2]https://bitbucket.org/smejkapa/cog-ai/src/master/Thesis/UserDocumentation.docx [Accessed 15.07.2022]

The program also outputted different files, like .gv files which contained graphs of the trees built by the different MCTS variants and logs that detailed messages and warnings encountered during the running of the program, but we didn't use these in our work.

Besides these files containing our data, we also had to work with xml files which detailed the settings of our algorithms and tests. These were stored in subfolders of a folder named Resources, present in the same directory as the executable. As these files were desinged by Pavel Šmejkal, their description can be found in his user documentation. Please note that, since he called the files that we used for our test settings "benchmarks", we will use the terms "benchmark" and "setting" interchangeably here. We will also use the term "benchmark set" to denote an xml file which contains multiple tests that are supposed to be run one after another.

A.2 XML Manager

This is a C++ program for creating and working with xml files that contain settings for the different AIs, benchmarks and benchmark sets. The program uses a configuration file (simply called config.txt) where it stores paths to two folders - one which contains subfolders with xml files (referred to as the *home directory*) and another which contains C# files that define the individual AIs (referred to as the *scripts directory*).

The home directory needs to contain subdirectories called AI, Benchmarks and BenchmarkSets for the program to accept it.

When the program starts, it checks if this file is present and if the paths contained in it are valid. If not, it prompts the user to enter paths to these directories. Otherwise, the user is free to enter commands. If the program detects that the user has entered the wrong number of arguments or that the arguments do not have the correct format, it informs the user and terminates the execution of the command.

The following subsections detail the commands that this program recognizes. At the end of each subsection is an example of the correct usage of the discussed command.

A.2.1 makeai

This is a command for creating xml files that correspond to AIs. Such a file is supposed to contain the parameters that are to be passed to the AI's constructor. The command has the following arguments:

- The name of the C# script from which the xml file is supposed to be created. This parameter is mandatory and has to be entered first, the others are optional and can be entered in arbitrary order. The name of the script needs to be entered without the ".cs" suffix.
- A list of arguments from the constructor of the class contained in the passed C# file that are supposed to be omitted in the xml file. The user starts the list by entering the "-e" option (e stands for exclude). The list can be terminated when another option is encountered, otherwise it continues until the end of the list of arguments. The default value of this option is an empty list.
- A list of C# data types that are allowed to be present in the xml file. The user starts the list by entering the "-p" option (p stands for permitted). The list can be terminated when another option is encountered, otherwise it continues until the end of the list of arguments. The default value of this option contains the types *int* and *double*.
- A list of names of scripts that the AI is supposed to have in its portfolio. The user starts the list by entering the "-s" option (s stands for scripts). The list can be terminated when another option is encountered, otherwise it continues until the end of the list of arguments. The default value of this option is an empty list.

If the program does not find the entered C# script in the scripts directory, the program informs the user and terminates the execution of the command. If everything goes smoothly, the program prompts the user to enter the values of all the parameters that are to be included in the xml file and then the name of the file. When entering the name, the program checks whether the name is allowed and whether a file with the same name does not already exist - if either of these conditions is false, the program communicates it to the user. The user can enter a different name (or choose to replace the present files) and the process is repeated. After the file is created, the programs notifies the user of success.

Example of the correct usage of this command:
makeai MyAIFile -s NOKAV Kiter -e undesirableArgName

A.2.2 makebmrk

This command creates an xml file that corresponds to a benchmark made from two AI files. As arguments, it takes the names of the two xml files that correspond

to the AIs. The names are entered without the ".xml" suffix. If one of the given files does not exist, the program notifies the user. If both files exist, the program tries to create a file called "name of the first file_vs_name of the second file". If such a file already exists, the program notifies the users and asks whether it is supposed to overwrite the file. If not, the user can enter a new name. The new name is checked for validity and again for conflicts with other files (if there is a conflict, the procedure repeats).

Example of the correct usage of this command:
makebmrk AIFile1 AIFile2

A.2.3 makebmrks

A command that gets a list of regular expressions as its arguments, then goes through the folder with AIs, picks those that match the regular expressions and creates benchmark files for all possible pairs. The regexes are supposed to be entered without the ".xml" suffix.

Example of the correct usage of this command:
makebmrks MCTS_(.*) Sigmoid_MCTS(.*)

A.2.4 makebset

This command serves to create a file that contains a set of benchmarks that are supposed to be run one after another. As its arguments, this command takes a list of regular expressions that are then matched to files in the directory which contains benchmarks. The regexes are supposed to be entered without the ".xml" suffix. After the command is entered, the program asks the user what the new file should be called. The entered name is checked for validity and conflict with existing files. If such a file already exists, the program notifies the users and asks whether it is supposed to overwrite the file. If not, the user can enter a new name. The user also enters a new name if the first one was found to be invalid. The new name is again checked and the procedure repeats.

Example of the correct usage of this command:
makebset (.*)MCTS Kiter

A.2.5 makebsets

A command that goes through all the xml files that specify AIs and finds all those that differ in more than just the number of playouts. From these files it

creates all the possible pairs, finds their corresponding benchmarks and creates a benchmark set from them. This command does not take any arguments.

Example of the correct usage of this command:
makebsets

A.2.6 delete

This command allows the user to delete files. It takes the following arguments:

- Location from which the files are supposed to be deleted (ai/benchmark/set depending on whether the files are supposed to be AI settings, benchmarks or benchmark sets). This argument has to be entered first.
- A list of regular expressions that get matched to files from the given folder. The matched files are deleted by the command.

After the command finishes, it informs the user of how many files were successfully deleted and how many failed.

Example of the correct usage of this command:
delete ai (.*)MCTS_100 Kiter_([0-9]*)

A.2.7 change

Serves to change attributes and elements of the xml files. It takes the following arguments in the order in which they are listed here:

- Location of the files that are supposed to be changed (ai/benchmark/set depending on whether the files are supposed to be AI settings, benchmarks or benchmark sets).
- The name of the element/attribute that is supposed to be changed.
- The new value of the element/attribute.
- Specification of whether it is an attribute or an element.
- The position of the occurrence of the element/attribute that is supposed to be changed. This is necessary because some files have some elements or attributes mentioned more than once.
- An arbitrary number of regular expression that are supposed to be matched to files that are to be changed.

Example of the correct usage of this command:
change benchmark Script Kiter elem 3 (.*)MCTS

A.2.8 sethome

Changes the path to the home directory. It takes just one argument - the new path (entered without quotes).

Example of the correct usage of this command:
sethome C:\Users\user_name\C++ programs

A.2.9 setscripts

This command changes the path to the scripts directory. It takes only one argument - the new path (entered without quotes).

Example of the correct usage of this command:
setscripts C:\Users\user_name\C# programs

A.2.10 quit

Terminates the program. Does not take any arguments. Just like other commands, it can not be entered while another command is being executed (for example, if the user types "quit" when prompted to enter some data by another command, "quit" just gets interpreted as data) with one exception - when the program asks the user to enter the paths to the home and scripts directories at the start. In this case, the program will terminate when the user types "quit" and presses enter.

Example of the correct usage of this command:
quit

A.2.11 help

Displays a help text which lists all the possible commands along with short descriptions. Does not take any arguments.

Example of the correct usage of this command:
help

A.3 CSV Cruncher

This program was made in C# and used to process the csv tables that were outputted by the benchmarks. Its main method goes through a list of csv files stored in a variable called *files*, extracts the relevant data from them (for us, that was the number of wins, the number of symwins, the remaining hull and the damage dealt to the enemy) and stores it in a variable called *sheets*. All the csv files should be in a single directory, the path to which is stored in a variable called *path*.

This variable keeps the values separated based on battle type and based on which implementation was going up against which. This allows us to process the data in various ways. We can, for example, create a table that averages the wins of all the variants over all the battles at 100 playouts. We can just as easily create a different table that averages the remaining hull of all the variants in a given battle setting over all the playout settings.

After the data is processed, a method can be called to output some tables. There are two main methods for this - *MakeCSVTable* and *MakeLatexTable* which output a csv file and a text file respectively, which contains the LaTeX definition of a table. Both of these methods make use of the *MakeSheet* method which is used to create a table which averages some of the data stored in *sheets*. Which data is supposed to be taken into account is determined through the method's parameters.

There is one more important method - *MakeAllLatexTables*. This method was made to print all the tables that are present in the appendix of the thesis.

A.4 Data Visualizer

This Python projects contains five Python scripts for processing different data and one script with auxiliary functions. Most of the scripts here visualize data in the form of graphs. These scripts always output multiple graphs in one image. The program doesn't support creating individual graphs - for our thesis, they had to be separated in a different program.

A.4.1 battledata1.py

Creates three bar graphs that show the average number of wins, symwins, remaining hull and damage dealt to the enemy for every variant. The graphs show data for three different battle settings - 4 vs 4, 8 vs 8 and 16 vs 16. There is also a fourth plot which copies the data obtained in the 16 vs 16 setting. The reason for its presence is that the other Python scripts we used for creating graphs make

four plots, so this ensures that they all have the same size.

The graphs are created using matplotlib and seaborn using data contained in files the path to which is stored in the *files* variable. They are sorted according to the field stored in the *sorting_field* variable. Each metric has a divisor associated with it (in the form of a variable named {*metric name*}*_divisor*) by which its values are divided. The divisors for wins and symwins are set to 1 while the divisors for hull and damage are set to $6 \cdot 3 = 18$ - this is the total number of battles that any variant plays against any other variant with a given combat setting (it is the number of repeats for a combat setting (6) times the number of playout settings (3)). These divisors are then further mutiplied when computing data for every plot by the number of units at one teams disposal in the given combat setting.

A.4.2 battledata2.py

A script that is just like battledata1.py, except it shows graphs for the three remaining combat settings (32 vs 32, 48 vs 48 and 64 vs 64) and the last graph shows data that is averaged over all the combat settings.

A.4.3 battlerankings.py

Loads the same data that was visualized in battledata1.py and battledata2.py, as well as data about the time and depth of the different variants, orders it and prints the rankings of the variants in a format that can be copied straight into LaTeX tables.

A.4.4 common_functions.py

An auxiliary script containing some functions used by multiple other scripts for fetching and processing data.

A.4.5 playoutdata.py

Like battledata1.py and battledata2.py, except it processes data that is grouped based on the number of playouts. Like in battledata2.py, the fourth plot shows an average of the data in the previous three plots.

The divisors for wins and symwins are again set to 1 while the divisors for hull and damage are set to $(4+8+16+32+48+64) \cdot 6 = 1032$ - this is the total number of units in all the battles that a single variant fought against any other variant with a given playout setting.

A.4.6 playoutrankings.py

Like battlerankings.py, except it ranks variants variants according to data accumulated over the number of playouts.

A.4.7 timedepthdata.py

Similarly to battledata1.py, battledata2.py and playoutdata.py, this script creates four bar graphs, this time detailing the average iteration time and average tree depth of the variants however. Whether this is done based on data accumulated over playout settings or combat settings can be configured by changing the files that the script loads.

This script also contains the *sorting_field* variable which determines which field is used to sort the graphs and this time also a variable called *ascending* which determines whether the ordering is supposed to be ascending or descending (this is due to the fact that, unlike the other metrics, time is supposed to be sorted in descending order, since less time is better).

A.5 Position Generator

A simple Python program that generates random positions for a given number of units of type battleship and destroyer around a given center. It is used to create randomized battle settings. The number of battleships is stored in the variable *num_of_b* and the number of destroyers in *num_of_d*. Variables *centerQ* and *centerR* contain the coordinates of the center point. The coordinates system used here is the axial coordinate system[3]. The variable *div_factor* stores a value by which the total number of units is supposed to be divided in the computation of the coordinates. Setting this number to a higher value makes the positions move closer to the center point. The positions are written out to the standard output in the format ' <Unit Id="{unit_type}" Q="{q_coordinate}" R="{r_coordinate}" />'. This is so that they can be copied to the xml files that contain battle settings without any further processing.

A.6 Result Trimmer

Trims the output of the benchmarking program so that, for every benchmark, only the names of the two variants, the data accumulated during the running of the benchmark (the number of wins, symwins and the remaining hull for both variants), the number of symdraws (these occur when both agents finish with the same amount of hull in two symmetric battles) - if there were any - and the total

[3]https://www.redblobgames.com/grids/hexagons/#coordinates-axial [Accessed 15.07.2022]

benchmark time are left. It has a list of files that it is supposed to process which is stored in the *inputFiles* variable. The complete path to every file is stored in every iteration of the main for cycle in the *input* variable. The progam processes these files one by one and outputs the results into a file, the path to which is stored in the *output* variable. Note that the StreamWriter that is used to write into the output file has the *append* flag set to true, which means that it will append to the output file instead of overwriting it.

A.7 Time-Depth Cruncher

This program processes the files which contain data about execution time and tree depth of the MCTS variants. It has a list of regexes that it stores in the *regex* variable and uses these to select specific files. It then computes the mean iteration time and tree depth from the data in those files, as well as their respective standard errors, and stores them in csv files with one line for each MCTS variant. The names of the MCTS variants are taken from the names of the folders in which the files are stored.

The paths to the input and output files are entered directly into the commands that create their respective writers and readers.

B Additional Data

This section contains tables with more detailed measurements than those presented in section 7. Every table contains measurements about one of the primary metrics for individual pairs of MCTS variants. Unlike in section 7, the data presented here is summed, not averaged.

wins (100 playouts)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	27	27	28	26	27	28	28	27	23
HP	9	X	20	25	19	19	20	19	19	13
QB	9	16	X	17	17	20	19	22	18	10
RB	8	11	19	X	14	18	18	19	14	12
Sig	10	17	19	22	X	17	16	19	18	13
SR	9	17	16	18	19	X	16	20	18	10
UCT	8	16	17	18	20	20	X	15	22	14
U-T	8	17	14	17	17	16	21	X	16	12
VOI	9	17	18	22	18	18	14	20	X	10
WP	13	23	26	24	23	26	22	24	26	X
wins (500 playouts)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	30	24	25	29	25	23	28	25	21
HP	6	X	19	19	23	18	20	20	18	14
QB	12	17	X	16	17	19	19	16	15	14
RB	11	17	20	X	14	17	14	16	16	14
Sig	7	13	19	22	X	16	23	16	15	12
SR	11	18	17	19	20	X	24	21	17	14
UCT	13	16	17	22	13	12	X	19	16	15
U-T	8	16	20	20	20	15	17	X	19	9
VOI	11	18	21	20	21	19	20	17	X	17
WP	15	22	22	22	24	22	21	27	19	X
wins (1000 playouts)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	26	29	29	29	27	28	31	28	23
HP	10	X	21	21	21	16	19	22	18	13
QB	7	15	X	17	18	15	17	17	12	12
RB	7	15	19	X	13	14	19	15	13	11
Sig	7	15	18	23	X	17	23	19	15	12
SR	9	20	21	22	19	X	21	21	18	16
UCT	8	17	19	17	13	15	X	19	14	11
U-T	5	14	19	21	17	15	17	X	16	12
VOI	8	18	24	23	21	18	22	20	X	13
WP	13	23	24	25	24	20	25	24	23	X

Table 1: The number of wins that every variant scored against every other variant on every playout setting.

symwins (100 playouts)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	12	17	17	12	18	18	17	18	12
HP	6	X	16	18	12	16	14	14	14	4
QB	1	2	X	6	6	9	5	7	5	0
RB	1	0	8	X	2	7	6	7	3	0
Sig	6	5	12	16	X	14	14	15	14	5
SR	0	2	5	7	4	X	6	9	6	0
UCT	0	4	6	7	4	7	X	5	6	3
U-T	1	4	4	6	3	4	9	X	5	2
VOI	0	4	13	15	4	12	12	13	X	2
WP	6	14	18	18	13	18	15	16	16	X
symwins (500 playouts)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	14	16	16	15	16	16	16	16	12
HP	4	X	15	15	15	18	17	17	15	4
QB	2	3	X	7	3	7	7	7	6	1
RB	2	3	8	X	2	7	7	6	4	1
Sig	3	2	15	16	X	14	16	15	13	1
SR	2	0	8	8	4	X	10	13	5	0
UCT	2	1	8	8	1	4	X	6	2	1
U-T	2	1	8	9	3	2	9	X	5	0
VOI	2	3	12	14	5	13	16	12	X	1
WP	6	14	17	17	17	18	17	18	17	X
symwins (1000 playouts)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	13	18	18	13	17	17	16	15	9
HP	5	X	16	17	14	15	15	18	16	1
QB	0	2	X	6	2	8	6	8	0	0
RB	0	1	9	X	2	4	11	5	4	0
Sig	5	4	16	16	X	15	17	15	11	1
SR	1	3	7	11	3	X	9	12	3	0
UCT	1	3	9	4	1	6	X	8	3	0
U-T	2	0	7	10	3	3	7	X	5	0
VOI	3	2	18	14	7	15	15	12	X	0
WP	9	17	18	18	17	18	18	18	18	X

Table 2: The number of symwins that every variant scored against every other variant on every playout setting.

hull (100 playouts)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	2491.1	2936.0	2981.9	2682.4	3067.7	2992.1	3034.1	2936.4	2271.9
HP	420.5	X	1644.3	1703.0	1437.4	1581.4	1640.4	1441.5	1525.2	1136.5
QB	306.6	1120.6	X	1338.8	1177.2	1471.1	1490.7	1575.3	1384.1	1017.4
RB	288.8	1149.3	1401.0	X	1076.1	1379.7	1412.2	1324.3	1320.6	1007.2
Sig	408.9	1215.2	1530.3	1670.8	X	1554.1	1439.1	1535.2	1500.5	1125.3
SR	356.1	1129.3	1390.3	1399.4	1212.8	X	1436.5	1571.3	1469.2	1037.9
UCT	290.7	1068.8	1428.4	1546.7	1257.6	1412.4	X	1298.0	1520.9	1075.1
U-T	264.2	1154.7	1367.9	1380.6	1200.7	1325.8	1453.4	X	1344.7	1012.5
VOI	331.8	1172.2	1506.7	1533.3	1220.2	1550.6	1367.0	1502.2	X	1043.1
WP	521.0	1408.2	1999.0	1841.0	1656.3	1932.1	1759.5	1759.8	1806.1	X
hull (500 playouts)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	2563.7	2974.7	2988.1	3004.0	2984.1	2946.6	3018.2	3043.1	2252.4
HP	213.0	X	1629.9	1629.9	1708.4	1930.8	1893.2	1815.4	1668.3	1173.1
QB	103.4	1038.1	X	1260.0	1086.1	1434.3	1336.9	1355.5	1311.9	1030.7
RB	115.0	1038.1	1396.8	X	1060.7	1395.4	1255.0	1205.9	1320.1	1030.7
Sig	149.8	1237.8	1614.0	1735.0	X	1657.1	1807.2	1612.0	1434.6	1143.7
SR	123.2	1135.9	1587.3	1614.7	1277.4	X	1785.3	1727.1	1492.0	1091.1
UCT	86.9	1041.5	1331.8	1463.0	1060.3	1262.1	X	1334.2	1310.3	1064.1
U-T	113.0	1034.0	1327.7	1314.0	1122.7	1354.6	1290.0	X	1450.1	1035.6
VOI	197.1	1087.3	1691.2	1790.8	1291.9	1739.6	1735.2	1548.4	X	1111.4
WP	421.4	1570.1	1984.3	1984.3	2126.6	2155.5	2088.3	2023.8	1921.6	X
hull (1000 playouts)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	2778.6	3670.2	3670.2	3465.9	3800.4	3534.9	3641.1	3631.2	2452.4
HP	197.7	X	1893.9	2057.2	1751.9	1918.0	1942.0	2161.4	1781.3	1268.5
QB	35.9	1055.4	X	1240.2	1090.2	1440.8	1323.2	1270.1	1276.2	986.1
RB	35.9	1033.6	1397.6	X	1086.4	1327.6	1434.1	1268.6	1284.1	1001.0
Sig	215.7	1306.8	1647.4	1958.1	X	1758.0	1950.5	1688.2	1570.4	1187.6
SR	112.4	1112.2	1673.9	1737.7	1136.2	X	1872.4	1803.1	1684.7	1127.5
UCT	74.2	1028.8	1273.4	1255.7	1060.2	1398.1	X	1350.5	1393.2	989.8
U-T	60.5	1029.5	1321.2	1301.8	1065.6	1346.0	1306.6	X	1461.0	1032.0
VOI	189.9	1149.5	1972.4	1835.0	1154.0	1874.8	1887.6	1853.5	X	1015.9
WP	425.4	1851.5	2319.7	2447.0	2026.3	2198.6	2505.7	2401.3	2144.9	X

Table 3: The amount of remaining hull that every variant scored against every other variant on every playout setting.

damage (100 playouts)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	7706.5	7820.4	7838.2	7718.1	7770.9	7836.3	7862.8	7795.2	7606.0
HP	5635.9	X	7006.4	6977.7	6911.8	6997.7	7058.2	6972.3	6954.8	6718.8
QB	5191.0	6482.7	X	6726.0	6596.7	6736.7	6698.6	6759.1	6620.3	6128.0
RB	5145.1	6424.0	6788.2	X	6456.2	6727.6	6580.3	6746.4	6593.7	6286.0
Sig	5444.6	6689.6	6949.8	7050.9	X	6914.2	6869.4	6926.3	6906.8	6470.7
SR	5059.3	6545.6	6655.9	6747.3	6572.9	X	6714.6	6801.2	6576.4	6194.9
UCT	5134.9	6486.6	6636.3	6714.8	6687.9	6690.5	X	6673.6	6760.0	6367.5
U-T	5092.9	6685.5	6551.7	6802.7	6591.8	6555.7	6829.0	X	6624.8	6367.2
VOI	5190.6	6601.8	6742.9	6806.4	6626.5	6657.8	6606.1	6782.3	X	6320.9
WP	5855.1	6990.5	7109.6	7119.8	7001.7	7089.1	7051.9	7114.5	7083.9	X
damage (500 playouts)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	7914.0	8023.6	8012.0	7977.2	8003.8	8040.1	8014.0	7929.9	7705.6
HP	5563.3	X	7088.9	7088.9	6889.2	6991.1	7085.5	7093.0	7039.7	6556.9
QB	5152.3	6497.1	X	6730.2	6513.0	6539.7	6795.2	6799.3	6435.8	6142.7
RB	5138.9	6497.1	6867.0	X	6392.0	6512.3	6664.0	6813.0	6336.2	6142.7
Sig	5123.0	6418.6	7040.9	7066.3	X	6849.6	7066.7	7004.3	6835.1	6000.4
SR	5142.9	6196.2	6692.7	6731.6	6469.9	X	6864.9	6772.4	6387.4	5971.5
UCT	5180.4	6233.8	6790.1	6872.0	6319.8	6341.7	X	6837.0	6391.8	6038.7
U-T	5108.8	6311.6	6771.5	6921.1	6515.0	6399.9	6792.8	X	6578.6	6103.2
VOI	5083.9	6458.7	6815.1	6806.9	6692.4	6635.0	6816.7	6676.9	X	6205.4
WP	5874.6	6953.9	7096.3	7096.3	6983.3	7035.9	7062.9	7091.4	7015.6	X
damage (1000 playouts)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	7929.3	8091.1	8091.1	7911.3	8014.6	8052.8	8066.5	7937.1	7701.6
HP	5348.4	X	7071.6	7093.4	6820.2	7014.8	7098.2	7097.5	6977.5	6275.5
QB	4456.8	6233.1	X	6729.4	6479.6	6453.1	6853.6	6805.8	6154.6	5807.3
RB	4456.8	6069.8	6886.8	X	6168.9	6389.3	6871.3	6825.2	6292.0	5680.0
Sig	4661.1	6375.1	7036.8	7040.6	X	6990.8	7066.8	7061.4	6973.0	6100.7
SR	4326.6	6209.0	6686.2	6799.4	6369.0	X	6728.9	6781.0	6252.2	5928.4
UCT	4592.1	6185.0	6803.8	6692.9	6176.5	6254.6	X	6820.4	6239.4	5621.3
U-T	4485.9	5965.6	6856.9	6858.4	6438.8	6323.9	6776.5	X	6273.5	5725.7
VOI	4495.8	6345.7	6850.8	6842.9	6556.6	6442.3	6733.8	6666.0	X	5982.1
WP	5674.6	6858.5	7140.9	7126.0	6939.4	6999.5	7137.2	7095.0	7111.1	X

Table 4: The amount of damage dealt to the enemy that every variant scored against every other variant on every playout setting.

wins (4 vs 4)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	9	9	9	9	10	8	9	9	7
HP	9	X	9	10	9	9	9	10	9	9
QB	9	9	X	9	10	9	9	9	9	9
RB	9	8	9	X	9	9	9	9	10	9
Sig	9	9	8	9	X	9	9	9	9	8
SR	8	9	9	9	9	X	9	8	9	8
UCT	10	9	9	9	9	9	X	9	9	9
U-T	9	8	9	9	9	10	9	X	9	8
VOI	9	9	9	8	9	9	9	9	X	8
WP	11	9	9	9	10	10	9	10	10	X

wins (8 vs 8)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	11	10	11	11	6	10	13	8	6
HP	7	X	9	12	13	6	12	10	7	4
QB	8	9	X	9	9	7	11	9	2	1
RB	7	6	9	X	6	5	12	6	3	2
Sig	7	5	9	12	X	6	14	11	6	3
SR	12	12	11	13	12	X	12	12	10	6
UCT	8	6	7	6	4	6	X	7	5	4
U-T	5	8	9	12	7	6	11	X	4	4
VOI	10	11	16	15	12	8	13	14	X	5
WP	12	14	17	16	15	12	14	14	13	X

wins (16 vs 16)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	18	15	16	18	16	15	16	16	17
HP	0	X	14	11	12	10	11	12	12	7
QB	3	4	X	8	11	9	10	10	7	6
RB	2	7	10	X	3	10	7	7	6	6
Sig	0	6	7	15	X	9	8	6	11	6
SR	2	8	9	8	9	X	9	14	8	3
UCT	3	7	8	11	10	9	X	10	9	6
U-T	2	6	8	11	12	4	8	X	9	3
VOI	2	6	11	12	7	10	9	9	X	9
WP	1	11	12	12	12	15	12	15	9	X

Table 5: The number of wins that every variant scored against every other variant in the first three combat settings.

wins (32 vs 32)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	12	12	12	12	13	12	13	14	9
HP	6	X	10	12	8	9	8	10	11	7
QB	6	8	X	8	7	11	10	8	10	5
RB	6	6	10	X	7	10	6	9	8	8
Sig	6	10	11	11	X	8	11	10	9	7
SR	5	9	7	8	10	X	12	10	9	8
UCT	6	10	8	12	7	6	X	8	10	7
U-T	5	8	10	9	8	8	10	X	11	6
VOI	4	7	8	10	9	9	8	7	X	6
WP	9	11	13	10	11	10	11	12	12	X

wins (48 vs 48)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	15	16	16	17	16	17	18	16	13
HP	3	X	9	11	12	10	10	10	7	4
QB	2	9	X	7	6	9	6	10	8	6
RB	2	7	11	X	7	6	8	10	7	3
Sig	1	6	12	11	X	9	11	9	4	4
SR	2	8	9	12	9	X	10	9	8	6
UCT	1	8	12	10	7	8	X	10	10	5
U-T	0	8	8	8	9	9	8	X	9	3
VOI	2	11	10	11	14	10	8	9	X	3
WP	5	14	12	15	14	12	13	15	15	X

wins (64 vs 64)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	18	18	18	17	18	17	18	17	15
HP	0	X	9	9	9	9	9	9	9	9
QB	0	9	X	9	9	9	9	9	9	9
RB	0	9	9	X	9	9	9	9	9	9
Sig	1	9	9	9	X	9	9	9	9	9
SR	0	9	9	9	9	X	9	9	9	9
UCT	1	9	9	9	9	9	X	9	9	9
U-T	0	9	9	9	9	9	9	X	9	9
VOI	1	9	9	9	9	9	9	9	X	9
WP	3	9	9	9	9	9	9	9	9	X

Table 6: The number of wins that every variant scored against every other variant in the second three combat settings.

symwins (4 vs 4)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	0	8	7	0	9	8	6	8	0
HP	9	X	9	9	3	9	9	9	9	4
QB	1	0	X	4	1	6	2	3	2	0
RB	2	0	5	X	0	5	7	4	7	0
Sig	9	4	8	9	X	9	9	9	9	4
SR	0	0	3	4	0	X	3	4	3	0
UCT	1	0	6	1	0	5	X	8	2	0
U-T	3	0	4	4	0	5	1	X	4	0
VOI	1	0	7	2	0	6	7	3	X	0
WP	9	5	9	9	5	9	9	9	9	X
symwins (8 vs 8)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	3	7	8	4	6	7	7	5	0
HP	6	X	7	8	7	6	8	6	4	0
QB	2	2	X	5	1	4	7	4	1	0
RB	1	1	4	X	1	3	6	4	0	0
Sig	5	2	8	8	X	4	8	8	5	1
SR	3	3	5	6	5	X	6	8	4	0
UCT	2	1	2	3	1	3	X	1	1	1
U-T	2	3	5	5	1	1	8	X	2	1
VOI	4	5	8	9	4	5	8	7	X	0
WP	9	9	9	9	8	9	8	8	9	X
symwins (16 vs 16)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	9	9	9	9	9	9	9	9	9
HP	0	X	8	7	8	9	6	9	8	3
QB	0	1	X	3	4	5	3	5	2	1
RB	0	2	5	X	0	3	4	3	2	1
Sig	0	1	5	9	X	6	6	4	8	1
SR	0	0	4	6	3	X	5	9	3	0
UCT	0	3	6	5	3	4	X	5	3	3
U-T	0	0	4	6	5	0	4	X	4	0
VOI	0	1	7	7	1	6	6	5	X	3
WP	0	6	8	8	8	9	6	9	6	X

Table 7: The number of symwins that every variant scored against every other variant in the first three combat settings.

symwins (32 vs 32)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	9	9	9	9	9	9	9	9	6
HP	0	X	8	9	7	9	7	9	9	1
QB	0	1	X	3	1	5	5	5	3	0
RB	0	0	6	X	0	3	4	5	1	0
Sig	0	2	8	9	X	7	9	9	9	0
SR	0	0	4	6	2	X	8	7	1	0
UCT	0	2	4	5	0	1	X	2	1	0
U-T	0	0	4	4	0	2	7	X	2	1
VOI	0	0	6	8	0	8	8	7	X	0
WP	3	8	9	9	9	9	9	8	9	X

symwins (48 vs 48)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	9	9	9	9	9	9	9	9	9
HP	0	X	6	8	8	8	7	7	6	1
QB	0	3	X	4	2	4	1	5	3	0
RB	0	1	5	X	1	4	3	2	1	0
Sig	0	1	7	8	X	8	8	6	2	0
SR	0	1	4	4	1	X	3	6	3	0
UCT	0	2	5	5	1	4	X	3	4	0
U-T	0	2	2	6	3	1	5	X	3	0
VOI	0	3	6	8	7	6	5	6	X	0
WP	0	8	9	9	9	9	9	9	9	X

symwins (64 vs 64)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	9	9	9	9	9	9	9	9	9
HP	0	X	9	9	8	8	9	9	9	0
QB	0	0	X	0	2	0	0	0	0	0
RB	0	0	0	X	4	0	0	0	0	0
Sig	0	1	7	5	X	9	7	9	5	1
SR	0	1	0	0	0	X	0	0	0	0
UCT	0	0	0	0	1	0	X	0	0	0
U-T	0	0	0	0	0	0	0	X	0	0
VOI	0	0	9	9	4	9	9	9	X	0
WP	0	9	9	9	8	9	9	9	9	X

Table 8: The number of symwins that every variant scored against every other variant in the second three combat settings.

hull (4 vs 4)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	113.4	180.8	193.3	101.0	361.2	263.4	245.2	302.1	68.3
HP	292.2	X	363.5	382.5	180.5	386.7	358.1	360.9	352.8	188.2
QB	103.2	30.4	X	230.1	30.0	278.0	243.5	262.2	276.7	54.2
RB	103.0	29.7	227.4	X	44.7	301.7	268.1	162.4	312.9	78.8
Sig	279.0	183.7	317.5	332.4	X	384.1	358.5	358.9	325.6	173.0
SR	151.3	54.9	210.8	188.3	58.7	X	249.5	223.3	319.5	89.9
UCT	80.7	28.6	307.9	203.9	21.5	305.3	X	251.2	270.4	71.0
U-T	72.4	26.1	261.4	201.6	36.5	310.3	150.3	X	279.3	40.9
VOI	173.0	49.4	312.7	196.8	48.5	345.8	276.6	230.9	X	41.6
WP	302.8	230.5	365.4	352.6	217.0	368.2	365.7	349.7	335.4	X

hull (8 vs 8)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	145.6	297.0	358.0	156.0	207.1	290.6	488.9	247.9	29.4
HP	200.3	X	303.6	497.0	360.8	279.8	613.8	535.4	226.5	83.4
QB	58.7	40.4	X	126.7	43.6	151.4	206.7	100.7	18.4	0.5
RB	65.2	13.9	202.7	X	20.3	89.2	213.7	88.8	41.7	6.7
Sig	184.8	97.3	322.3	482.7	X	188.5	463.9	415.8	188.5	19.5
SR	168.4	105.3	363.7	418.7	161.7	X	581.6	534.8	244.6	50.1
UCT	74.3	10.6	75.5	145.0	16.9	56.5	X	58.7	61.5	19.5
U-T	78.7	42.4	110.8	119.5	41.1	122.1	211.8	X	53.9	45.2
VOI	254.9	131.8	485.2	461.6	176.5	325.3	534.9	446.0	X	46.0
WP	523.3	379.6	829.8	828.3	564.0	594.2	848.2	645.8	563.5	X

hull (16 vs 16)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	653.5	1194.5	1196.2	1257.8	1288.6	1180.3	1163.3	1132.1	492.3
HP	0.0	X	539.6	356.7	335.5	471.1	418.8	403.2	480.3	164.0
QB	31.6	23.5	X	156.3	98.7	260.1	269.3	243.2	105.7	40.9
RB	24.2	41.2	235.6	X	6.5	208.9	184.8	110.0	139.0	38.4
Sig	0.0	69.8	257.0	605.3	X	378.1	235.7	168.0	379.6	118.4
SR	13.8	42.7	288.8	230.8	66.9	X	332.6	448.5	318.3	75.3
UCT	48.0	46.5	191.3	295.0	76.3	196.2	X	195.7	260.0	44.2
U-T	42.7	13.9	83.9	203.9	80.0	27.1	123.5	X	268.6	1.8
VOI	27.5	59.9	354.3	383.0	68.8	391.9	297.9	345.7	X	69.1
WP	1.1	313.5	559.7	541.2	552.7	710.4	490.9	509.3	347.2	X

Table 9: The amount of remaining hull that every variant scored against every other variant in the first three combat settings.

hull (32 vs 32)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	598.6	1116.6	1115.0	1040.6	1149.2	999.7	991.0	1166.8	559.7
HP	299.5	X	512.6	575.3	460.2	569.3	553.5	600.9	549.2	194.4
QB	250.8	21.0	X	75.5	15.1	286.4	115.4	158.3	200.7	13.1
RB	245.7	12.0	136.8	X	32.1	169.5	158.1	165.3	174.8	12.2
Sig	283.9	194.1	412.4	415.2	X	463.3	505.0	431.2	377.3	149.8
SR	237.7	96.1	398.6	408.0	109.5	X	530.9	438.0	272.3	93.3
UCT	243.4	25.7	73.1	184.1	16.2	122.3	X	188.6	266.7	42.6
U-T	243.9	16.3	144.5	141.2	43.8	211.9	239.5	X	282.1	41.7
VOI	240.0	58.2	449.3	513.3	101.8	482.1	522.8	485.4	X	56.5
WP	374.6	448.5	707.1	688.3	631.5	742.6	763.5	788.8	788.5	X

hull (48 vs 48)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	2284.8	2547.5	2556.7	2456.9	2615.7	2533.3	2584.9	2569.6	2020.9
HP	39.2	X	339.3	469.1	472.3	613.8	421.9	408.4	281.4	66.1
QB	1.6	102.1	X	140.9	67.8	260.8	206.4	327.0	286.1	30.2
RB	1.6	106.9	283.4	X	15.0	223.9	167.1	162.8	171.8	22.4
Sig	22.7	199.6	373.0	418.8	X	445.7	524.2	352.0	149.9	70.2
SR	20.5	63.5	280.1	396.5	146.3	X	290.1	347.4	406.6	41.5
UCT	3.3	41.8	276.3	327.9	150.6	282.8	X	179.0	281.2	34.1
U-T	0.0	121.0	306.7	220.7	110.7	245.5	215.4	X	287.3	45.5
VOI	13.3	120.5	459.3	494.9	188.5	510.4	248.1	286.6	X	57.7
WP	125.7	453.3	731.5	752.4	744.5	761.3	775.7	781.8	753.4	X

hull (64 vs 64)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	4037.5	4244.5	4221.0	4140.0	4230.4	4206.3	4220.1	4192.2	3806.1
HP	0.0	X	3109.5	3109.5	3088.4	3109.5	3109.5	3109.5	3084.6	2882.0
QB	0.0	2996.7	X	3109.5	3098.3	3109.5	3109.5	3109.5	3084.6	2895.3
RB	0.0	3017.3	3109.5	X	3104.6	3109.5	3109.5	3109.5	3084.6	2880.4
Sig	4.0	3015.3	3109.5	3109.5	X	3109.5	3109.5	3109.5	3084.6	2925.7
SR	0.0	3014.9	3109.5	3109.5	3083.3	X	3109.5	3109.5	3084.6	2906.4
UCT	2.1	2985.9	3109.5	3109.5	3096.6	3109.5	X	3109.5	3084.6	2917.6
U-T	0.0	2998.5	3109.5	3109.5	3076.9	3109.5	3109.5	X	3084.6	2905.0
VOI	10.1	2989.2	3109.5	3109.5	3082.0	3109.5	3109.5	3109.5	X	2899.5
WP	40.3	3004.4	3109.5	3109.5	3099.5	3109.5	3109.5	3109.5	3084.6	X

Table 10: The amount of remaining hull that every variant scored against every other variant in the second three combat settings.

damage (4 vs 4)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	274.8	463.8	464.0	288.0	415.7	486.3	494.6	394.0	264.2
HP	453.6	X	536.6	537.3	383.3	512.1	538.4	540.9	517.6	336.5
QB	386.2	203.5	X	339.6	249.5	356.2	259.1	305.6	254.3	201.6
RB	373.7	184.5	336.9	X	234.6	378.7	363.1	365.4	370.2	214.4
Sig	466.0	386.5	537.0	522.3	X	508.3	545.5	530.5	518.5	350.0
SR	205.8	180.3	289.0	265.3	182.9	X	261.7	256.7	221.2	198.8
UCT	303.6	208.9	323.5	298.9	208.5	317.5	X	416.7	290.4	201.3
U-T	321.8	206.1	304.8	404.6	208.1	343.7	315.8	X	336.1	217.3
VOI	264.9	214.2	290.3	254.1	241.4	247.5	296.6	287.7	X	231.6
WP	498.7	378.8	512.8	488.2	394.0	477.1	496.0	526.1	525.4	X

damage (8 vs 8)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	933.7	1075.3	1068.8	949.2	965.6	1059.7	1055.3	879.1	610.7
HP	988.4	X	1093.6	1120.1	1036.7	1028.7	1123.4	1091.6	1002.2	754.4
QB	837.0	830.4	X	931.3	811.7	770.3	1058.5	1023.2	648.8	304.2
RB	776.0	637.0	1007.3	X	651.3	715.3	989.0	1014.5	672.4	305.7
Sig	978.0	773.2	1090.4	1113.7	X	972.3	1117.1	1092.9	957.5	570.0
SR	926.9	854.2	982.6	1044.8	945.5	X	1077.5	1011.9	808.7	539.8
UCT	843.4	520.2	927.3	920.3	670.1	552.4	X	922.2	599.1	285.8
U-T	645.1	598.6	1033.3	1045.2	718.2	599.2	1075.3	X	688.0	488.2
VOI	886.1	907.5	1115.6	1092.3	945.5	889.4	1072.5	1080.1	X	570.5
WP	1104.6	1050.6	1133.5	1127.3	1114.5	1083.9	1114.5	1088.8	1088.0	X

damage (16 vs 16)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	2268.0	2236.4	2243.8	2268.0	2254.2	2220.0	2225.3	2240.5	2266.9
HP	1614.5	X	2244.5	2226.8	2198.2	2225.3	2221.5	2254.1	2208.1	1954.5
QB	1073.5	1728.4	X	2032.4	2011.0	1979.2	2076.7	2184.1	1913.7	1708.3
RB	1071.8	1911.3	2111.7	X	1662.7	2037.2	1973.0	2064.1	1885.0	1726.8
Sig	1010.2	1932.5	2169.3	2261.5	X	2201.1	2191.7	2188.0	2199.2	1715.3
SR	979.4	1796.9	2007.9	2059.1	1889.9	X	2071.8	2240.9	1876.1	1557.6
UCT	1087.7	1849.2	1998.7	2083.2	2032.3	1935.4	X	2144.5	1970.1	1777.1
U-T	1104.7	1864.8	2024.8	2158.0	2100.0	1819.5	2072.3	X	1922.3	1758.7
VOI	1135.9	1787.7	2162.3	2129.0	1888.4	1949.7	2008.0	1999.4	X	1920.8
WP	1775.7	2104.0	2227.1	2229.6	2149.6	2192.7	2223.8	2266.2	2198.9	X

Table 11: The amount of damage dealt to the enemy that every variant scored against every other variant in the first three combat settings.

damage (32 vs 32)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	4236.5	4285.2	4290.3	4252.1	4298.3	4292.6	4292.1	4296.0	4161.4
HP	3937.4	X	4515.0	4524.0	4341.9	4439.9	4510.3	4519.7	4477.8	4087.5
QB	3419.4	4023.4	X	4399.2	4123.6	4137.4	4462.9	4391.5	4086.7	3828.9
RB	3421.0	3960.7	4460.5	X	4120.8	4128.0	4351.9	4394.8	4022.7	3847.7
Sig	3495.4	4075.8	4520.9	4503.9	X	4426.5	4519.8	4492.2	4434.2	3904.5
SR	3386.8	3966.7	4249.6	4366.5	4072.7	X	4413.7	4324.1	4053.9	3793.4
UCT	3536.3	3982.5	4420.6	4377.9	4031.0	4005.1	X	4296.5	4013.2	3772.5
U-T	3545.0	3935.1	4377.7	4370.7	4104.8	4098.0	4347.4	X	4050.6	3747.2
VOI	3369.2	3986.8	4335.3	4361.2	4158.7	4263.7	4269.3	4253.9	X	3747.5
WP	3976.3	4341.6	4522.9	4523.8	4386.2	4442.7	4493.4	4494.3	4479.5	X
damage (48 vs 48)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	6764.8	6802.4	6802.4	6781.3	6783.5	6800.7	6804.0	6790.7	6678.3
HP	4519.2	X	6701.9	6697.1	6604.4	6740.5	6762.2	6683.0	6683.5	6350.7
QB	4256.5	6464.7	X	6520.6	6431.0	6523.9	6527.7	6497.3	6344.7	6072.5
RB	4247.3	6334.9	6663.1	X	6385.2	6407.5	6476.1	6583.3	6309.1	6051.6
Sig	4347.1	6331.7	6736.2	6789.0	X	6657.7	6653.4	6693.3	6615.5	6059.5
SR	4188.3	6190.2	6543.2	6580.1	6358.3	X	6521.2	6558.5	6293.6	6042.7
UCT	4270.7	6382.1	6597.6	6636.9	6279.8	6513.9	X	6588.6	6555.9	6028.3
U-T	4219.1	6395.6	6477.0	6641.2	6452.0	6456.6	6625.0	X	6517.4	6022.2
VOI	4234.4	6522.6	6517.9	6632.2	6654.1	6397.4	6522.8	6516.7	X	6050.6
WP	4783.1	6737.9	6773.8	6781.6	6733.8	6762.5	6769.9	6758.5	6746.3	X
damage (64 vs 64)										
	FAP	HP	QB	RB	Sig	SR	UCT	U-T	VOI	WP
FAP	X	9072.0	9072.0	9072.0	9068.0	9072.0	9069.9	9072.0	9061.9	9031.7
HP	5034.5	X	6075.3	6054.7	6056.7	6057.1	6086.1	6073.5	6082.8	6067.6
QB	4827.5	5962.5	X	5962.5	5962.5	5962.5	5962.5	5962.5	5962.5	5962.5
RB	4851.0	5962.5	5962.5	X	5962.5	5962.5	5962.5	5962.5	5962.5	5962.5
Sig	4932.0	5983.6	5973.7	5967.4	X	5988.7	5975.4	5995.1	5990.0	5972.5
SR	4841.6	5962.5	5962.5	5962.5	5962.5	X	5962.5	5962.5	5962.5	5962.5
UCT	4865.7	5962.5	5962.5	5962.5	5962.5	5962.5	X	5962.5	5962.5	5962.5
U-T	4851.9	5962.5	5962.5	5962.5	5962.5	5962.5	5962.5	X	5962.5	5962.5
VOI	4879.8	5987.4	5987.4	5987.4	5987.4	5987.4	5987.4	5987.4	X	5987.4
WP	5265.9	6190.0	6176.7	6191.6	6146.3	6165.6	6154.4	6167.0	6172.5	X

Table 12: The amount of damage dealt to the enemy that every variant scored against every other variant in the second three combat settings.