TotalBotWar: An Innovative AI Challenge and Competition for Pseudo Real-time Multi-action games

Raúl Montoliu[1,*], Alejandro Estaben[1], César Díaz[1], Sergi Fuster[1] and Diego Pérez-Liebana[2]

[1]*Institute of New Imaging Technologies. Jaume I University. Castellón. Spain*
[2]*Game AI group. Queen Mary University of London. London, UK.*

Abstract

This paper introduces *TotalBotWar*, a novel pseudo-real-time multi-action challenge for game AI. Additionally, it includes initial experiments that assess the framework's performance with various agents. *TotalBotWar* is inspired by the real-time battles found in the popular TotalWar game series, where players command armies to defeat their opponents. In this game, each turn comprises a set of orders to control one's units. As the game progresses, the number and specific orders that can be executed in a turn change. An intriguing aspect of *TotalBotWar* is that if a unit doesn't receive an order in a turn, it continues the action it performed in the previous turn. This characteristic results in a rapidly increasing turn-wise branching factor, making it challenging for traditional algorithms. Furthermore, the game's partial observability of the game state makes it a valuable platform for testing modern AI algorithms.

Keywords

Game AI, Multi-action games, Bots programming

1. Introduction

In recent years, games have proven to be important test-beds for Artificial Intelligence (AI). For instance, deep reinforcement learning has enabled computers to learn how to play games such as Chess [1], Go [1], Atari games [2], and many other games [3]. Despite these important advances, there are still games that pose important challenges for state-of-the-art AI agents. Some examples are *Blood Bowl* [4], *Legend of Code and Magic* [5], *MicroRTS* [6], *FightingICE* [7], *Hanabi* [8], *Splendor* [9], *StarCraft* [10], and the *General Video Game AI* framework [11], among others.

In this paper, we propose *TotalBotWar*, a new pseudo-real-time challenge for game AI. The game is inspired by the real-time battles of the popular *TotalWar* game series[1], where two players control respective armies with the objective of defeating each other. On each turn, the agent must decide where the unit must move to. When two opposite units collide, they will start to fight. The result of the combat depends on the type of units and their attributes. If during a turn a unit does not receive any order, it will continue its movement following the

CEV'23. II Congreso Español de Videojuegos. Madrid, 9 y 10 de noviembre de 2023
*Corresponding author.
montoliu@uji.es (R. Montoliu); diego.perez@qmul.ac.uk (D. Pérez-Liebana)
0000-0002-8467-391X (R. Montoliu)

[1]Creative Assembly, https://www.totalwar.com/

previous one, or it will stand still if none was given. This introduces unknown information on the state: it is possible to know that an enemy unit is moving, but not its destination. The game has a high number of possible actions in a turn ($\approx 6.7E7$) and also a huge number of possible states ($\approx 3.3E29$), which provides a significant challenge for AI agents. The game, implemented using the *CodinGame* SDK[2], has already been made available online at this platform[3].

An initial set of experiments is also presented, where five different agents are benchmarked to give a baseline to future researchers. Three of them are primary agents where a) units never move (but can fight), b) always move forward, or c) move to a random localisation. The two remaining are more sophisticated. The first one applies human knowledge by using a heuristic function. The last one implements the Online Evolutionary planning (OEP) algorithm proposed by Justesen et al. [12]. Preliminary results show that the heuristic-based and OEP overcome the three primary agents, being the OEP preferable.

Summarising, this paper presents three main novelties:

- We present *TotalBotWar* a new pseudo-real-time multi-action challenge for game AI.
- As far as we know, this is the first work implementing real-time *TotalWar*-style battles as a game AI challenge.
- We assess the performance of five agents, including the Online Evolutionary Planning algorithm, in the proposed game AI challenge.

The rest of the paper has been organised as follows. Section 2 presents the main characteristics of the game including how agents interact with the game engine. A set of baseline agents and some preliminary experiments are shown in Sections 3 and 4. Finally, the most important conclusions drawn from this work are summarised in Section 5.

2. The game

2.1. Game Overview

TotalBotWar is a 1 vs 1, pseudo-real-time, multi-action game, partially inspired by the real-time battles of the *Total War* games series. In our game, both players start with the same number of military units and their objective is to defeat the other player. The winner is the player who first destroys all the opponent's units or the one with more units alive on the battlefield when the maximum number of turns is reached, which is set to 400. There are four different types of units: Swordsmen, Spearmen, Archers and Knights (see Figure 1). The game uses a classical *rock-paper-scissors* combat scheme, where swordsmen are good against spearmen, spearmen are good versus knights and finally, knights are good against swordsmen. Archers are an exception: they can attack from a distance but are very weak in face-to-face combat.

Each unit has an attribute vector modelling its behaviour. The attributes are Health Points, Attacking Strength, Defence, Charge Power, Charge Resistance, Moving Speed and defence against Arrows. Besides, archers also have Throwing Distance and Arrow Damage. Table 1 shows the values assigned to each attribute for each unit type.

[2]https://www.codingame.com/

[3]https://www.codingame.com/contribute/view/486222077fe22e3aa6bcdc0f729dd46223bb

Figure 1: The game units, from left to right: swordsmen, spearmen, archers and knights.

Table 1
Values of the attributes of each unit type.

Attribute	Swordsmen	spearmen	Knights	Archers
Health Points	250	250	200	100
Attacking Strength	20	15	12	10
Defence	10	20	12	5
Charge Power	5	10	100	5
Charge resistance	25	125	15	0
Moving Speed	15	10	40	15
Defence against Arrows	10	30	30	10
Throwing Distance	-	-	-	450
Arrow damage	-	-	-	20

Units can move to any place on the battlefield. Two units from the same agent can't overlap, and they will fight if they belong to different armies. If a unit reaches the limits of the battlefield, it stops. Archers always shoot arrows at enemy troops into the attacking range. Troops suffer friendly fire if they are close to an opponent unit receiving arrows.

The game has three different leagues or levels. When using the *CodinGame* platform, the player has to first implement a bot to defeat the system bot of the first league. After, he/she must implement a new one to defeat the system bot of the second league before passing to the third one. In the third league, the *CodingGame* platform allows testing the player's bot versus the bots implemented by many other players. Alternatively, the three leagues can be used isolated from *CodingGame* platform to test AI algorithms and to rule AI competitions.

In the first league (see Figure 2), the army of each player consists of just one unit of each type and units start in predefined initial locations on the battlefield. The second league introduces the draft phase (see Figure 3) where, in the first 9 turns of the game, the agents must select how many units of each type will be part of their armies and their initial positions on the battlefield. Therefore, the total number of units is 9. There is no restriction on the number of units of each type, i.e. the army can be composed of 9 archers if this is the decision of the agent. On each turn, players select units simultaneously, knowing only the units selected by both players in all previous turns. In the third league (see Figure 4), the army is composed of 30 units, therefore the draft lasts until the 30^{th} turn of the game. Additionally, the third league introduces the *General* unit, which is a highly important unit that can be crucial in the game. All units within a distance of 150 pixels from their general increase (multiplicatively) their attributes by 1.25.

Figure 2: Screenshot of the first league of the game. The army has 4 units, one of each type.

Figure 3: Screenshot of the second league of the game during the draft phase. In this league, the army is composed of 9 different units. The composition of the army is defined at the draft phase.

However, if the general is dead, all units decrease their attributes by 0.75. The general is always the first unit selected in the draft and can be of any type.

The size of the battlefield is 1920×1080 pixels. The battlefield is completely flat. The size of the units is 150×150 pixels in leagues 1 and 2, and 75×75 pixels in the third league.

2.2. Main characteristics

The main characteristics of game AI are as follows:

- It is a 1 vs 1 game.
- It is (pseudo) real-time. Although the game engine performs actions in the order indicated in the turn, the effect of this order is practically negligible. Similarly, the effect of which player performs the actions is minimal.

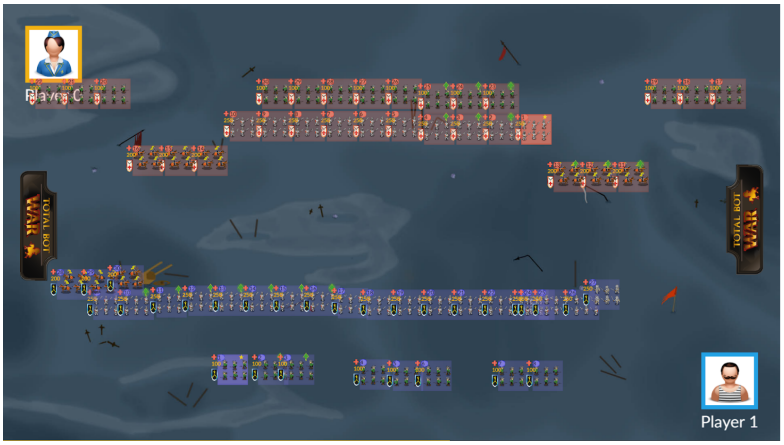

Figure 4: Screenshot of the third league of the game during the battle phase. The army is composed of 30 units and includes a General. The General is the unit with ID 1 and its background colour is highlighted.

- Not all information is known in the state. The state contains information about the actual position of the enemy units and if they are moving or not, but it does not provide information about the final target where they are moving.
- It is multi-action since in the same turn more than one action can be performed for each different unit owned on the battlefield.
- The agents have just 200ms to decide the actions to be executed on each turn. This is a restriction of the core of *CodingGame* engine.
- It has a very large number of possible actions in a turn ($\approx 6.7E7$) and possible states ($\approx 3.3E29$).

2.3. Motivation

This work has two principal motivations. On the one hand, the game has been included in *CodinGame* platform to be used as a tool to learn programming skills fascinatingly. *CodinGame* platform allows the use of many programming languages, and it is possible to see the effect of the source code used for the agent developed. That can help beginners to learn programming languages faster than through a more traditional style of teaching. The first league of the game is perfect for this purpose. On the other hand, the second and third leagues are dedicated to the development of new game AI agents due to their complexity.

The development of a new game AI challenge using the *CodinGame* SDK has three main advantages with respect to completely developing it from zero:

1. Developers can take advantage of the framework which contains useful code that can make it easier to develop a new game.
2. Users can program their agents in their preferred programming language instead of being restricted, as usually happens, to use just the one used to develop the game.

3. Sometimes starting with a new game AI challenge is hard since, for instance, users do not always have installed the correct libraries to run the game. The use of *CodinGame* platform avoids this kind of problem.

However, some constraints must be accomplished as the maximum size of the battlefield, the amount of *thinking* time per turn for the agents and the maximum number of turns, among others.

The game is inspired by *Total War* games since they are very popular with the general public. Similarly to other popular games AI challenges such as *StarCraft* AI competition [10], can engage students to learn programming languages in general and AI in particular since they can be highly motivated to develop agents to play popular games.

2.4. Action Space

On each turn, the current player can provide an action for each one of their units. An action consists of moving a unit a particular number of pixels in both x and y directions and the movement normally takes several turns to be completed. If in a turn the player does not indicate an action for a particular unit, it continues the movement following the previous action performed on this unit.

An action has the following format: "$ID\ \delta_x\ \delta_y$" where:

- ID is the unique ID of the unit.
- δ_x is the number of pixels we want to move the troop on the X axis.
- δ_y is the number of pixels we want to move the troop on the Y axis.

Note that δ_x and δ_y are not the global coordinates to move the unit to, but how many pixels the unit must move with respect to its current coordinates. The coordinates are relative to the unit location to be independent to the position of the agent (up or down) in the battlefield.

For instance, some actions that can be played are:

- "1 100 50": Unit with ID 1 will move 100 pixels to its right (east in the display if the agent plays in the bottom part of the battlefield or west, otherwise) and 50 to the front of the battlefield (upwards or north in the display if the agent plays in the bottom part of the battlefield or downwards or south, otherwise).
- "3 -100 -10": Unit with ID 3 will move 100 pixels to its left and 10 pixels backward.
- "5 0 0": Unit with ID 5 will stop.

On each turn, a player can perform more than one action using a string separated by semicolons. For instance, to perform the three previously described actions in the same turn, the player would provide the following multi-action string: "1 100 50; 3 -100 -10; 5 0 0".

2.5. State representation

The system provides information about the player and the opponent's units. First, the game indicates the total number of units for each player's army. Then, the system provides the following information for each one of the player and opponent's units:

Table 2
Number of possible actions in the draft phase depending on the battlefield size.

Battlefield size	# actions
1920×1080	$8.3E6$
26×14	$1.5E3$
13×7	$3.6E2$

Table 3
Number of possible actions in the battle phase depending on the league and the battlefield size.

Battlefield size	1^{st} league	2^{nd} league	3^{rd} league
1920×1080	$8.3E6$	$1.9E7$	$6.2E7$
26×14	$1.5E3$	$3.3E3$	$1.1E4$
13×7	$3.6E2$	$8.2E2$	$2.7E3$

- ID: Unique ID of the unit.
- Location: x, y coordinates indicating the actual position of the unit on the battlefield.
- Direction: a number indicating where the unit is looking for. It can be `Northwest` (0), `North` (1), `Northeast` (2), `East` (3), `Southeast` (4), `South` (5), `Southwest` (6) and `West` (7).
- Life: amount of health points (See Table 1). The unit is dead when its life reaches 0.
- Type: unit type for swordsmen (0), spearmen (1), cavalry (2) and archers (3).
- Moving: Indicates if the unit is moving (1) or not (0).
- Target: x, y coordinates indicating where the unit is going to stop, only for friendly units (for opponent units, no target information is provided).

Therefore, the state has $1 + 9n + 7n$ elements, where n is the number of units for each player's army.

2.6. Game complexity

The number of possible actions that can be played on each turn is huge in both the draft and battle phases, due to the large battlefield size (1920×1080). It also depends on the league, 1 to 3, selected. One possibility to handle its complexity is to artificially reduce the places where the units can be moved. According to the size of the units, we suggest defining two grids, the first one of 13×7 ($1920/150 \approx 13$, $1080/150 \approx 7$) and the second one of 26×14 ($1920/75 \approx 26$, $1080/75 \approx 14$). Note that the units can always be moved to any place on the battlefield. The use of the grid is just for reducing the complexity of the game and it must be handled by agent programmers. It is suggested to be used in the first stages of the implementation of the agent, or for beginners.

Tables 2 and 3 show the number of actions in both phases with respect to the size of the battlefield in the three sizes proposed: 1920×1080, 26×14 and 13×7. The number of actions in the draft phase depends on the size of the battlefield (H, W) and the existing number of unit types ($t = 4$). This number can be calculated as:

$$H \times W \times t \tag{1}$$

Table 4
Number of different army combinations that can be obtained in the draft phase depending on the league and the battlefield size.

Battlefield size	1^{st} league	2^{nd} league	3^{rd} league
1920×1080	1	$7.5E7$	$2.5E8$
26×14	1	$1.3E4$	$4.4E4$
13×7	1	$3.3E3$	$1.1E4$

Table 5
Number of possible states in the battle phase.

Battlefield size	1^{st} league	2^{nd} league	3^{rd} league
1920×1080	$5.8E27$	$3.0E28$	$3.3E29$
26×14	$3.2E16$	$7.1E16$	$2.4E17$
13×7	$4.9E14$	$2.5E15$	$2.8E16$

For instance, when the smallest grid is used ($H = 13, W = 7$), the number of actions in the draft phase is $13 * 7 * 4 = 3.6E2$.

The number of actions in the battle phase depends on the size of the battlefield (H, W) and the number of units in each league (n). n is 4, 9 and 30 in leagues 1, 2 and 3, respectively. The number of actions can be calculated as:

$$H \times W \times n \tag{2}$$

For instance, when the smallest grid is used ($H = 13, W = 7$) and for the third league ($n = 30$), the number of actions in the battle phase is $13 * 7 * 30 = 2.7E3$.

Table 4 shows the number of existing army combinations that can be obtained in each league. This number depends on the size of the battlefield (H, W), the existing number of unit types ($t = 4$), and the number of units in each league (n). In this case, the formula is:

$$H \times W \times t \times n \tag{3}$$

For instance, when the complete battlefield is used ($H = 1920, W = 1080$) and for the second league ($n = 9$), the number of actions in the battle phase is $1920 * 1080 * 4 * 9 = 7.5E7$. Note that in the first league, there is just one possible combination since there is no draft phase and the initial configuration of the army is always the same.

Table 5 shows the number of states for the three leagues and proposed battlefield sizes. This number depends on the size of the battlefield (H, W), the number of different directions ($d = 8$), the number of health points (l) (for simplicity, we assume in these calculations that all units have the same number of health points $l = 100$, see Table 1), the existing number of unit types ($t = 4$), if the unit is moving or not ($m = 2$), and the number of units on each league (n). The number of states can be calculated as:

$$(H \times W \times d \times l \times t \times m \times H \times W) \times n \times (H \times W \times d \times l \times t \times m) \times n \tag{4}$$

Figure 5: Some animations used in the game.

Note that the second term, corresponding to the opponent units, does not have a second element $H \times W$ since the final target of the opponent units is unknown. For instance, when the complete battlefield is used ($H = 1920, W = 1080$) and for the first league ($n = 4$), the number of actions in the battle phase is $(1920 \times 1080 \times 8 \times 100 \times 4 \times 2 \times 1920 \times 1080) \times 4 \times (1920 \times 1080 \times 8 \times 100 \times 4 \times 2) \times 4 = 5.8E27$.

2.7. Game Art

One of the more interesting features of *CodinGame* is that it is possible to replay the game. Therefore, it is possible to study how some actions have affected the game at a particular moment of the game. A set of assets has been designed for a better representation of the game. The units are based on the middle age and have a cartoon style (see Figure 1). The game also includes animations for each state in which each unit can be found. The states are: idle, running, attacking and dead. Furthermore, there is an animation for when a unit is under an arrow attack. As an example, Figure 5 shows the different sprites of some of the animations used in the game.

3. Baseline agents implemented

Several AI agents have been developed as baselines for the proposed game. They are briefly explained as follows:

3.1. Simple Agents

- StayStatic (*SS*): All units stand still during the battle. A predefined army is always selected in the draft. The knights are in the flanks, spearmen and swordsmen in the middle and archers behind. The units never move but they can fight when colliding with an opponent. Besides, the archers can shoot arrows at opponent troops into the attacking range.
- AlwaysForward (*AF*): All units always move forward. The predefined army is the same as in *SS*.
- Random (*RND*): All units select random destinations. The predefined army is the same as in *SS*.

3.2. Heuristic (Λ)

It uses human knowledge in both phases. In the draft, the agent tries to pick the unit to have an advantage against the opponent. For instance, if the opponent selected in the previous turn a Knight, it will pick a Spearman. The agent has some rules to avoid choosing too many units of the same type. The agent selects the position in front of the troop that can be defeated by the selected one.

For the battle, for each unit, a heuristic function λ is used to estimate a value indicating how good is to attack each enemy unit. The enemy unit with the biggest value is the one selected as the target. The heuristic function λ has been designed as the average of 5 factors ϕ_i ($\phi_i \in [0, \dots, 1], \forall i$ and $i \in [1, \dots, 5]$) as follows:

$$\lambda = \frac{\phi_1 + \phi_2 + \phi_3 + \phi_4 + \phi_5}{5} \tag{5}$$

where:

- ϕ_1 provides higher values if the player's unit belongs to a type with an advantage with respect to the opponent one, taking into account the rock-paper-scissor combat system. It can be 1.0, 0.5 or 0.0 when the opponent unit type is worse, the same or better, respectively.
- ϕ_2 is 1.0 if the player's unit avoids getting into the opponent archer attacking range; 0.0 otherwise.
- ϕ_3 benefits from having more health points than the opponent unit. It can be 1.0, 0.5 or 0.0 when the opponent unit has less, the same or more health points, respectively.
- ϕ_4 is 1.0 in case of a flank attack, i.e. the attacking direction is not frontal, and 0.0 otherwise.
- ϕ_5 will be higher the closer the player's unit is to the enemy's.

In the third league, a new factor ϕ_6 is added with a value of 1.0 if the opponent unit is a general and 0.0 otherwise.

3.3. Online Evolutionary Planning (OEP)

This algorithm, proposed by Justesen et al. [12], evolves a vector of N moves to be executed by agents in multi-action games. In the original algorithm, an initial population of vectors (individuals) is generated at random to then be evolved by the algorithm, executing actions consecutively in the forward model. The state reached when all actions are executed is evaluated to obtain fitness for the individual.

The OEP agent implemented for *TotalBotWar* uses the method described for the Heuristic agent for the draft phase and for seeding the initial population in the battle. Each individual contains N genes, where each gene corresponds to a unit owned by the agent and their values are the IDs of the opponent's unit to attack, i.e. the number of genes N is the number of units n of the army in each league. For instance, in the first league ($n = 4$), a genome $[2, 1, 0, 1]$ indicates that the first unit from the OEP agent will attack the opponent unit with ID 2, the second unit will attack unit with ID 1, and so on. A mutation rate $\rho = 0.1$ is applied to each gene to change the target to attack. The resultant states are evaluated using the same ϕ_i factors as in Λ, but adding a new one that rewards individuals who target the same opponent unit more than once.

Table 6
Win-rate of agents tested in League 3.

	SS	*AF*	*RND*	Λ	*OEP*
SS		0.0	0.2	0.0	0.0
AF	1.0		0.5	0.0	0.05
RND	0.8	0.5		0.2	0.1
Λ	1.0	1.0	0.8		0.5
OEP	1.0	0.95	0.90	0.5	

4. Experiments

Several games have been played using the agents developed and described in Section 3. These experiments have been performed outside the CodinGame platform. Table 6 shows the win rate of the agents playing as the first player. In all cases, results are reported using the third league, the complete battlefield size (i.e. no grid is used) and 100 games. As expected, Λ and *OEP* agents overcome the simplest baselines. Surprisingly there is a tie between Λ and *OEP* agents. This is likely due to the *OEP* agent not having enough time (with the time limit constrain of *CodingGame* platform) to evolve stronger action selections, being unable to find better recommendations than the ones initially provided by the Λ agent.

We have also tested both algorithms (*OEP* vs Λ) in leagues 2 and 1, obtaining a win rate for the *OEP* of 0.65 and 0.90, respectively. In these cases, the game is less complex than in the case of the league 3. Therefore, the *OEP* agent performs more iterations in the allowed budget time and it is able to find better move recommendations.

5. Conclusions

This paper presents a new game for game AI: *TotalBotWar*. The game introduces interesting features and challenges for AI, as it presents a pseudo-real-time decision-making problem in a large and continuous state and action space. It also provides an interesting challenge for drafting policies, army composition and tactical planning. The paper suggests different possibilities for discretizing the state space and also benchmarks a state-of-the-art algorithm, Online Evolutionary Planning (OEP), which shows good results in the simpler scenarios but can't outperform domain-knowledge rule-based agents in the complex ones due to the time limitation constrain in *CodingGame* platform.

Future work can span in multiple directions. Regarding the game, more complex units, rules and terrain features could be added. In terms of agents, sophisticated agents and techniques can be objects of research to outperform the proposed baselines. Finally, we plan to propose this benchmark as a new competition in the future for game-playing AI research.

References

[1] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, D. Hassabis, A general reinforcement

learning algorithm that masters chess, shogi, and go through self-play, Science 362 (2018) 1140–1144. doi:10.1126/science.aar6404.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, D. Hassabis, Human-level control through deep reinforcement learning, Nature 518 (2015) 529–533. doi:10.1038/nature14236.

[3] N. Justesen, P. Bontrager, J. Togelius, S. Risi, Deep learning for video game playing, IEEE Transactions on Games 12 (2017) 1–20. doi:10.1109/TG.2019.2896986.

[4] N. Justesen, L. M. Uth, C. Jakobsen, P. D. Moore, J. Togelius, S. Risi, Blood bowl: A new board game challenge and competition for ai, in: 2019 IEEE Conference on Games, 2019.

[5] J. Kowalski, R. Miernik, Legends of code and magic, https://jakubkowalski.tech/Projects/LOCM/, 2019. [Online; accessed 8-April-2020].

[6] S. Ontañón, N. A. Barriga, C. R. Silva, R. O. Moraes, L. H. S. Lelis, The first microrts artificial intelligence competition., AI Magazine 39 (2018) 75–83.

[7] R. Ishii, S. Ito, R. Thawonmas, T. Harada, A fighting game ai using highlight cues for generation of entertaining gameplay, in: 1st IEEE Conference on Games (CoG'19), 2019. doi:10.1109/CIG.2019.8848069.

[8] J. Walton-Rivers, P. R. Williams, R. Bartle, The 2018 hanabi competition, in: 2019 IEEE Conference on Games (CoG), IEEE, 2019, pp. 1–8.

[9] I. Bravi, D. Perez-Liebana, S. M. Lucas, J. Liu, Rinascimento: Optimising statistical forward planning agents for playing splendor, in: 2019 IEEE Conference on Games (CoG), IEEE, 2019, pp. 1–8.

[10] M. Čertický, D. Churchill, K. Kim, M. Čertický, R. Kelly, Starcraft ai competitions, bots, and tournament manager software, IEEE Transactions on Games 11 (2019) 227–237. doi:10.1109/TG.2018.2883499.

[11] D. Perez-Liebana, S. M. Lucas, R. D. Gaina, J. Togelius, A. Khalifa, J. Liu, General Video Game Artificial Intelligence, Morgan & Claypool Publishers, 2019. https://gaigresearch.github.io/gvgaibook/.

[12] N. Justesen, T. Mahlmann, S. Risi, J. Togelius, Playing multi-action adversarial games: Online evolutionary planning versus tree search, IEEE Transactions on Games 10 (2018) 281–291. doi:0.1109/TCIAIG.2017.2738156.