NTBOE: a new algorithm for efficiently playing multi-action adversarial games

Diego Villabrille[1], Raul Montoliu[1,*]

[1]Institute of New Imaging Technologies, Jaume I University. Castellón, Spain

Abstract

This paper presents N-Tuple Bandit Online Evolution (*NTBOE*), a novel algorithm for efficiently playing multi-action adversarial games. Its main advantage is the use of a structure composed of a set of multi-armed bandits to store information about the best combination of actions that can be played in the player's turn. Accessing this structure, to get an estimation of how good is to be in a particular game state, is several times faster than accessing the forward model of the game. Thanks to that, it is possible to explore more game states than the state of the art algorithms since it is able to go deeper in the search space. The performance of the proposed method has been assessed using the game *ASMACAG* as a testbed. The results obtained show that the proposed method overcomes state-of-the-art methods such as *Monte Carlo Tree Search* and *Online Evolution.*

Keywords

Artificial Intelligence for videogames, Multi action games, Evolutionary algorithms

1. Introduction

In turn-based multi-action adversarial games, each player turn consists of several actions and the order in which the agent plays those actions has a significant influence in the game. Evolutionary algorithms are the current state-of-the-art in this kind of games, overcoming other popular methods as Monte Carlo Tree Search [1].

Recently, the N-Tuple Bandit Evolutionary algorithm (*NTBEA*) [2] was presented as an effective method for parameter tuning. It is very useful when the evaluation function of the problem is noisy and fairly expensive in CPU time, as it uses to be the case in multi-action adversarial games. It allows exploring the space of possibilities choosing only the most promising options. Therefore, it is capable of reaching good, though not necessarily optimal, solutions in a very short time. This method has been used mainly in the field of artificial intelligence applied to games to optimize the heuristic function for a multi-action card game [3], optimize the hyperparameters of the agent that is capable of solving a set of games [4] and to model the player experience [5].

This paper presents a novel evolutionary algorithm which uses a set of multi arm bandits as in the *NTBEA* algorithm to store information of the best action combinations to be played in the player's turn. This new method to play multi-action adversarial games is called N-Tuple

I Congreso Español de Videojuegos, December 1–2, 2022, Madrid, Spain

[*] Corresponding author.

diego.villabrille@uji.es (D. Villabrille); montoliu@uji.es (R. Montoliu)
0000-0001-9176-2868 (D. Villabrille); 0000-0002-8467-391X (R. Montoliu)

Bandit Online Evolution (*NTBOE*). To assess the performance of the proposed method, the game *A Simple Multi-Action CArd Game* (*ASMACAG*) [6] has been used. It is a card game specially developed to be used as testbed for game artificial intelligence agents.

The main contributions of this work are as follows:

- We present a complete re-implementation of *ASMACAG* using Python 3.8 and a set of several good practices to provide code that is easily readable and understandable.
- We present N-Tuple Bandit Online Evolution (*NTBOE*), a new method to play multi-action adversarial games.
- We assess the performance of *NTBOE* playing versus two baseline methods (*Random* and *One Step Looking Ahead*) and two state-of-the-art ones (*Monte Carlo Tree Search* and *Online Evolution*).

As far as we know, this is the first paper using a method such as *NTBEA* to obtain the best combination of actions to be played in the player's turn.

The rest of the paper has been organized as follows: Section 2 presents a summary of the most relevant related methods. Section 3 exposes the problem definition that this paper tries to solve and Section 4 explains the *ASMACAG* game. Section 5 briefly explains the 4 methods implemented to be compared with the proposed one, presented in Section 6. The experiments done to assess the performance of the proposed method and the results obtained are showed in Section 7. Finally, Section 8 summarizes the main conclusions arisen from this work.

2. Related Work

One of the problems of previous multi action games' implementations is that, due to complex game rules, it should be quite complex to implement a bot for playing them. This is one of the reasons for developing *ASMACAG* since it provides a simple enough context in which the bots can be tested and understood and it allows for easy step by step debugging, while being complex enough that the results it yields are statistically relevant. For instance, *ASMACAG* is very convenient for artificial intelligence courses since the way to implement a new bot is very straightforward.

One of the most relevant works for playing multi action games is [7] where the authors first presented the *Online Evolution* (*OE*) method that overcomes *MTCS* implementation [1]. It is based on the concept of applying genetic algorithms by evolving actions during a game (online) instead of using evolution to train an agent that will play the game later (offline). This concept was first introduced, though applied to a different type of games, by the Rolling Horizon Evolution algorithm proposed at [8]. The Online Evolution algorithm tries to apply the aforementioned idea by using an evolutionary algorithm on a set of actions composing a turn. As it is stated by Niels Justesen in [7], "it can be seen as a single iteration of Rolling Horizon Evolution with a very short horizon (one turn)". The algorithm tries to find the best combination of actions for a turn by evolving an ordered set of actions. A more recent work in this game is *Evolutionary MCTS* for multi action adversarial games [9], which combines the tree search of *MCTS* with the sequence-based optimization of evolutionary algorithms, obtaining better results that OE in some particular conditions.

The main difference of the proposed *NTBOE* method with respect to the state of the art ones is that *NTBOE* stores information about the action combinations that can be used for selecting new turn candidates to be analyzed. In addition, evaluating the candidates using the internal structure composed of multi armed bandits is several times faster than evaluating using the forward model of the game. Therefore, using *NTBOE* more game states can be explored in the same time period increasing the probability of finding the optimum actions combination.

3. Definitions, Notation and Problem Formulation

This section introduces the terminology and the notation that will be used throughout this paper and formulates the problem to be solved. The notation follows the one exposed by P. Cowling et al. in [10]. More detail on the game theory concepts can be found in standard textbooks on this subject, e.g. [11].

A *game* is defined as a direct graph (S, Λ) where S are the nodes and Λ the edges of the graph. The nodes S are called *states* of the game. The leaf nodes are called *terminal states* and the other nodes *non-terminal states.* In general, a game has a positive number of k players. Some games also have an *environment player* (player 0). Each state s is associated with a number $\rho(s) \in \{0, \dots, k\}$, that represents the player about to act. Each terminal state s_T is associated with a vector $\mu(s_T) \in \mathbb{R}^k$, which represents the *reward* vector. In some games, the non-terminal states can also be associated with an *immediate* reward vector that gives an idea of how well the players are doing. A heuristic function $\lambda(s)$ can be used to estimate the reward vector $\mu(s)$, given the state (terminal or not).

The game starts at time $t = 0$ in the initial state s_0. At time $t = 0, 1, 2, \dots$, if state s_t is non terminal, player $\rho(s_t)$ chooses an edge $(s_t, s_{t+1}) \in \Lambda$ and the game transitions through that edge to the state s_{t+1}. This continues until a terminal state is reached at time $t = T$. Then, each player receives a reward equal to their corresponding entry in the vector $\mu(s_T)$ and the game ends. If the game allows immediate rewards, players receive the immediate reward $\mu(s_{t+1})$ after reaching the state s_{t+1}.

Players typically do not choose edges directly, but choose *actions.* In the simplest case, each outgoing edge of a state s corresponds to an action that player $\rho(s)$ can play. The set of actions from a state s is denoted $A(s)$. Note that, given a state s, the player $\rho(s)$ can play just one action (i.e. choose an edge (s_t, s_{t+1})) from the ones included in $A(s)$.

In multi-action games a player can play several consecutive actions in a *turn.* A turn, given a state s_t at time t, is defined as a list $\tau(s_t) = [a_1, a_2, \dots, a_q]$ where q is the number of consecutive actions that the player must play in its turn, $a_1 \in A(s_t)$, $a_2 \in A(s_{t+1})$ and $a_q \in A(s_{t+q-1})$. The order in which actions are played into the turn can be very important on some multi-action games, since to play some action before other can modify the immediate reward obtained after playing the last action of the turn a_q.

Therefore, the problem that the agents have to solve is, given a game state s_t, to find the list $\tau(s_t)$ that leads to the highest immediate expected reward $\tau(s_{t+q})$ when the turn finished (i.e. after playing the last action a_q of the turn).

4. A Simple Multi-Action CArd Game (*ASMACAG*)

ASMACAG is a simple card game proposed as a tool to test, develop and debug bots that implement artificial intelligence algorithms in the context of multi-action games. It was first presented in [6] using *Java* programming language. For this work, the game has been completely reimplemented using *Python 3.8* programming language. It has been implemented using several best practices such as the use of SOLID principles, PEP8 conventions, type hinting, and the production of a valuable documentation using docstrings. The aim of this is to have a code that is easily readable, understandable and that allows the user of *ASMACAG* to quickly understand anything they could need to. The documentation of the game can be access in https://dvs99.github.io/ASMACAG/index.html.

4.1. Game Rules

The rules for *ASMACAG* are simple by design, since the game has been designed as a tool whose aim is to be as straightforward as possible. This design philosophy allows to easily step-by-step debug the bots and understand their decisions while also staying complex enough for the results to be relevant and representative of their performance. The rules in the base version are as follows (note that each player of the game is a bot following an specific algorithm for decision-making):

- It is an adversarial game with 2 players, each of them is dealt 9 cards.
- There is a board with 20 cards dealt on it, which must contain only numbers.
- The cards include numbers from *1* to *6* and the special values *x2* and *%2*. The deck where the card are dealt from contains 8 cards of each of the numbers from *2* to *5*, 5 cards of numbers *1* and *6* and 6 instances of each one of the special cards.
- Each player must play 3 cards per turn with no possibility to skip. Playing a card consists in taking it from your hand and placing it on any card available on the board, if it is numbered, or just using it directly if it has an special value. After playing a card both the played card and the used card from the board, if applicable, will be discarded. Turns alternate until there are no cards left on either player's hand.
- Each time a numbered card is played $(P - B) * F$ points are awarded to the player that used it, where P is the value of the card played, B is the value of the card on the board and F is a factor that defaults to 1. Special cards affect the value of F when computing the score of the next numbered card played by any of the players. The *x2* special card will duplicate F while the *%2* card will halve it.

Some examples are as follows:

- The player plays a 6 card on a 2 card. $F = 1$. The players gets $(6 - 2) * 1 = 4$ points.
- The player plays a 6 card on a 2 card. $F = 2$. The players gets $(6 - 2) * 2 = 8$ points.
- The player plays a 4 card on a 6 card. $F = 2$. The players gets $(4 - 6) * 2 = -4$ points.
- The player plays a 5 card on a 2 card. $F = 0.25$ since, e.g. the opponent played two consecutive *%2* cards in the previous turn. The players gets $(5 - 2) * 0.25 = 0.75$ points.

An aspect taken into account in the design of the *ASMACAG* software is the need for flexibility in the test environment, specially regarding the rules. All these rules need to be programmed to be easily modified as needed during the testing process, by just coding a class that implements a forward model compatible with the game. Also, the parameters of the rules need to be easily accessible and changeable. This parameters include things, such as the number of cards dealt at each time, the amount of existing cards of each type, the range of numbers in the numbered cards, the number of actions to play per turn, etc.

According to the terminology exposed in the Section 3, in *ASMACAG* the number of players is $k = 2$, there is not environment player and $q = 3$, i.e. the turn $\tau(s_t)$ is composed by three actions. Thanks to the existence of the special cards *x2* and *%2*, the order in which the actions are played can modify the immediate reward $\mu(s_{t+3})$ obtained after playing the actions contained in the turn $\tau(s_t)$.

4.2. How to implement a new agent

Implementing a new bot for playing *ASMACAG* is very straightforward. A new bot must inherit from *Player* class and implement three methods: the constructor, the one who returns a string with the name of the bot and the *think* function, which given the current state of the game and a time budget must return the action to be played. As an example, the implementation of a bot that always returns a random action from the existing ones given the current game state of the game can be seen in https://github.com/dvs99/ASMACAG/blob/master/Players/RandomPlayer.py.

5. Baseline and state of the art methods

5.1. Random

This algorithm will just choose randomly one of the available valid actions when required to decide, without looking into the next action to play at all. It is included to be used as a minimal baseline, to check that each of the other algorithms are able to beat it in an statistically relevant and consistent way.

5.2. Greedy One-Step Lookahead

This algorithm will just perform a greedy search on the available valid actions at one moment of the turn, playing the one that returns the best value. Once again, it will not go through the next actions that could be played during that turn. It is useful as a baseline heuristic algorithm, testing the capability of each of the other algorithms to win against it, when applying the same heuristic.

5.3. Monte Carlo Tree Search Algorithm

This algorithm is probably the most researched of the ones tested and it has been selected for its relevance and importance in the AI field during the last decades. First coined in 2006 [12], it has been proposed almost since its conception as a useful algorithm for game AI [13]. It iteratively builds a tree that estimates the long term value of each action until the time budget is hit. Then it returns the estimated best performing action at that point. The nodes of the tree represent the state space of the game, while the edges represent the actions. Therefore, a child node of a given node represents the game state of the game after playing one of the possible actions that can be played from this game state. Each node a stores two values: s_a the average of the rewards that have been obtained in the past by choosing this node (i.e. playing the action), and n_a the number of times that this node has been chosen.

At each iteration, the method has to selected which child node has to be explored using a multi arm bandit. This is made by the UCB equation which balance between exploiting the node with the best $\overline{s_a}$ or exploring nodes with less visits n_a. The formula UCB for a particular node a is expressed as follows:

$$UCB(a) = \overline{s_a} + c\sqrt{\frac{\ln N}{n_a + \epsilon}} \tag{1}$$

$\overline{s_a}$ is the average of the rewards that have been obtained in the past by choosing the action a, N is the total number of times an action has had to be chosen, n_a is the number of times the action a has been chosen, and C is a parameter which balances between the exploitation and the exploration parts of the UCB functions. Greater values benefits exploration versus exploitation. The ϵ value is a small number (e.g. $\epsilon = 0.05$) that is used to avoid having to divide by zero in the equation. In those cases, i.e. when an action has never been chosen before, the value obtained will be very high to give the opportunity to explore, at least once, all possible actions.

In our particular implementation of the MCTS algorithm, the tree only explores the possible action combination in a player turn.

5.4. Online Evolution Algorithm

This algorithm has been selected for its relevance for the problem of multi-action adversarial games and focuses on the idea of solving the problem that is the most specific to multi-action games, which is finding the right combination of actions to compose a turn and therefore should be quite efficient when trying to win at *ASMACAG*. The algorithm has the followings steps:

1. A set of N_P individuals are randomly generated. Each individual (or genome) is a combination of actions that can be played in a turn. In *ASMACAG* each individual has three components (or genes).
2. The set of individuals are evaluated using the forward model and a score is obtained using a heuristic function.
3. The worst individuals, according to the obtained score, are removed from the set. This process is controlled by the survivor rate α parameter.
4. New individuals are generated doing crossover operations.
5. Each individual is mutated according to a mutation rate β.
6. The process is repeated until convergence or until the time budget is reached.

6. N-Tuple Bandit Online Evolution

As a novel approach to solving multi-action adversarial games, the N-Tuple Bandit Evolutionary Algorithm(NTBOE) is proposed. This algorithm was first designed for its use in offline optimization of an heuristic function [4] and of for tuning the hyperparameters of an agent for solving a set of games [3]. The idea suggested is attempting to apply it to evolving in-game actions during the turn calculation, using a similar approach to the one that *OE* uses to adapt traditional evolutionary algorithms.

The N-Tuple model in the algorithm directly models the statistics, while approximating the fitness and number of evaluations of each modeled combination of parameters [2]. This way, the algorithm places the focus on the combinations of the set of parameters being optimized. Applying it to multi-action decision making, the aim is to further focus on finding the right combinations of actions. Since this is the main problem of multi-action games, as discussed before, this approach could give interesting results for this specific kind of game.

The advantage it can have over other approaches, and specifically against other evolutionary algorithms, is that the N-Tuple based model in the algorithm is faster to access than the game itself. The algorithm tests several turns against that model, which approximates the reward based on the data received up to that point, per each time it tests a turn with the actual game. This allows the *NTBOE* algorithm to test a lot more possible turns in the same amount of time.

Algorithm 1 shows the procedure of the method. The first step of the model is create the N-Tuple Bandits model Γ where the score obtained after playing the actions into the turn, using the forward model, will be stored. In this work, a 1D Multi arm bandit per action (i.e. 3) is used together with 3 2D multi arm bandits for action combination (1st action + 2nd action, 1st action + 3rd action, and 2nd action + 3rd action). In total, the model consists of 6 multi arm bandits.

The second step of the method is an initialization step. It is detailed in Algorithm 2. A population Π of N_p random individuals is obtained and each one is evaluated using the forward model of the game (line 6). The model Γ is updated with the scores obtained in these evaluations. The best individual (and its score) is returned as the current one. This process is very important to fill the model with valuable information. Then, the iterative process stars, at each iteration N_n new individuals are created mutating the current one (line 5). For mutating an individual one action is randomly selected (using β) and it is randomly changed for another possible action given the state of the game at this moment. We call to this new individuals *neighbors*. Each neighbor is evaluated using the model Γ (line 10) and a score is obtained. Note that at this moment the individual is evaluated using the model Γ instead of the forward model. These individuals are evaluated using the model Γ by the Equation 2. It consist of the sum of all the UCB values (see Equation 1) of each one of the bandits belonging to the model. Evaluating an individual using Γ is several times faster than evaluating it using the forward model.

$$T_{UCB}(\tau(s_t)) = \sum_{b\in\Gamma} UCB(b, \tau(s_t)) \tag{2}$$

The previous process returns a candidate to be a better solution to the current one. It is evaluated using the forward model and the model Γ is updated with the score obtained. Then, if its score is better than the one obtained by the current, it becomes the new current. If not, the current individual remains being the same.

Algorithm 1 Proposed N-TBOE algorithm

```
**Require:** $N_p \in N^+$: Number of initial evaluations
**Require:** $N_n \in N^+$: Number of neighbors
**Require:** $N_i \in N^+$: Number of iterations
**Require:** $\beta \in [0, 1]$: Mutation probability
**Ensure:** $actions$: best action combination obtained
$\Gamma \leftarrow CreateModel()$
$current, current\_score \leftarrow Initialization(N_p, \Gamma)$
$i \leftarrow 0$
**while** $i < N_i$ **do**
    $\Pi \leftarrow GetNeighbors(current, N_n, \beta)$
    $j \leftarrow 0$
    $best\_score \leftarrow -\infty$
    $best\_neighbor \leftarrow None$
    **while** $j < N_n$ **do**
        $score \leftarrow EvaluateWithModel(\Pi(j), \Gamma)$
        **if** $score > best\_score$ **then**
            $best\_score \leftarrow score$
            $best\_neighbor \leftarrow \Pi(j)$
        **end if**
        $j \leftarrow j + 1$
    **end while**
    $score \leftarrow EvaluateWithForwardModel(best\_neighbor)$
    $UpdateModel(best\_neighbor, score, \Gamma)$
    **if** $score > current\_score$ **then**
        $current\_score \leftarrow score$
        $current \leftarrow best\_neighbor$
    **end if**
    $i \leftarrow i + 1$
**end while**
$actions \leftarrow current$
```

At the end, the current individual is the actions combination (or turn) $\tau(s_t)$ obtained as solution.

7. Experiments and Results

7.1. Experimental set-up

The performance of the proposed *NTBOE* method is assessed playing games against the two baselines and the two state of the art methods presented in Section 5. In all cases 1000 games have been played between two methods. Each algorithm will be the first player for half of the games played against any other algorithm and the second player for the other half of them.

Algorithm 2 Initialization of the *N-TBOE* algorithm

```
Require: $N_p \in N^+$: Number of initial evaluations
Require: $\Gamma$: N-Tuple Bandits Model
Ensure: $current$: best action combination evaluated
Ensure: $current\_score$: score of the best action combination
$\Pi \leftarrow RandomPopulation(N_p)$
$j \leftarrow 0$
$current\_score \leftarrow -\infty$
$current \leftarrow None$
while $j < N_p$ do
    $score \leftarrow EvaluateWithForwardModel(\Pi(j))$
    $UpdateModel(\Pi(j), score, \Gamma)$
    if $score > current\_score$ then
        $current\_score \leftarrow score$
        $current \leftarrow \Pi(j)$
    end if
    $j \leftarrow j + 1$
end while
```

This will allow to reduce the bias introduced by the fact that the game is easier to win as the first player. All algorithms will have a budget to make decisions which will be defined as time limit per turn, this time will be of 1 second. Regarding the evaluation of the game states where an heuristic $\lambda(s_t)$ is needed, all bots will make use of the same one, it being the score difference between the players. Finally, to evaluate the performance of each bot the number of wins will be used.

Two experiments have been performed:

1. The best parameter combination is obtained for bots *MCTS*, *OE* and *NTBOE*. For each of this bots a set of possible values per parameter will be chosen. The other parameters are fixed to a particular values. Then, 1000 games will be played against *OSLA* Player per each possible value for the parameter, keeping the rest of the parameters constant. Table 1 shows the configuration of this experiments.
2. An all versus all tournament has been performed among all the tuned versions of the bots. 1000 games have been played between two methods.

7.2. Results

Table 2 shows the results of the experiment performed to tune the parameter C in *MCTS*. The optimum value is $C = 8$. Table 3 shows the results of the experiments done for tuning the parameters of the *OE* algorithm. According to the results obtained the parameters for this method has been adjusted to $N_p = 125$, $\alpha = 0.15$, $\beta = 0.15$. Table 4 shows the results of the experiments done for tuning the parameters of the proposed *NTBOE* algorithm. According to

Table 1
Bot to be tuned, parameter to be tuned, possible values and how other parameters have been fixed.

Bot	Parameter	Values	Other parameters
MCTS	C	$\{0.5, 1.414, 3, 8, 14, 20, 30, 45\}$	-
OE	N_p	$\{25, 75, 125, 175\}$	$\alpha = 0.15, \beta = 0.35$
OE	β	$\{0.05, 0.15, 0.25, 0.35\}$	$N_p = 75, \alpha = 0.35$
OE	α	$\{0.05, 0.15, 0.35, 0.55, 0.75\}$	$N_p = 75, \beta = 0.15$
NTBOE	N_n	$\{5, 10, 20, 50, 120\}$	$\beta = 0.3, N_p = 500$
NTBOE	β	$\{0.05, 0.15, 0.3, 0.55, 0.8\}$	$N_n \in 20, N_p \in 500$
NTBOE	N_p	$\{50, 100, 500, 1000, 3000\}$	$N_n = 20, \beta = 0.3$

Table 2
Results table from the experiment for adjusting *MCTS* Player's C value. Best result in bold.

C value	*MCTS wins*	*OSLA wins*	*Ties*
0.5	502	485	13
0,141	509	473	18
3	523	458	19
8	**545**	**436**	**19**
14	520	470	10
20	507	478	15
30	477	507	16
45	458	530	12

the results obtained the parameters for this method has been adjusted to $N_n = 5$, $\beta = 0.55$, $N_p = 1000$.

Figure 1 shows the results of the second experiment. It can be seen that *OSLA* Player is the least performant bot, which is the expected result given that it doesn't make any attempt on finding a good combination of actions for the whole turn. Then *MCTS* Player follows relatively closely, showing that it can find combinations of actions but it is not as efficient as the evolutionary algorithms in doing so. Then both OE Player and *NTBOE* Player are quite ahead, because they use their evolutionary aspect to very efficiently find good combinations of actions. Within them, *NTBOE* seems to have a not extremely big but clearly significant advantage. In the particular match against *OE* the proposed *NTBOE* method was able to won the 60% of the games. This proves that this novel approach to multi-action game solving, proposed on this work, can yield better results than the current state-of-the-art algorithms.

The experiments presented in this work can be easily reproduced by running the Pyhton script https://github.com/dvs99/ASMACAG/blob/master/reproduce_experiments.py.

8. Conclusions

This paper has presented a novel method called N-Tuple Bandit Online Evolution to play multi action adversarial games. The results obtained have shown that the proposed *NTBOE* algorithm

Table 3
Results table from the experiment for adjusting OE Player's N_p, β and α parameters. Best result in bold.

N_p	β	α	*OE wins*	*OSLA wins*	*Ties*
25	0.15	0.35	669	314	17
75	0.15	0.35	695	292	13
125	**0.15**	**0.35**	**704**	**286**	**10**
175	0.15	0.35	692	292	16
75	0.05	0.35	680	302	18
75	**0.15**	**0.35**	**695**	**292**	**13**
75	0.25	0.35	678	303	19
75	0.35	0.35	675	308	17
75	0.15	0.05	688	302	10
75	**0.15**	**0.15**	**700**	**283**	**17**
75	0.15	0.35	695	292	13
75	0.15	0.55	671	306	23
75	0.15	0.75	675	311	14

Table 4
Results table from the experiment for adjusting *NTBOE* N_n, β and N_p parameters. Best result in bold.

N_n	β	N_p	*NTBEA wins*	*OSLA wins*	*Ties*
5	**0.3**	**500**	**733**	**250**	**17**
10	0.3	500	733	254	13
20	0.3	500	721	268	11
50	0.3	500	668	309	23
120	0.3	500	569	410	21
20	0.05	500	730	248	22
20	0.15	500	731	257	12
20	0.3	500	721	268	11
20	**0.55**	**500**	**772**	**216**	**12**
20	0.8	500	727	254	19
20	0.3	50	736	245	19
20	0.3	100	733	249	18
20	0.3	500	721	268	11
20	**0.3**	**1 000**	**744**	**244**	**12**
20	0.3	3 000	720	266	14

overcomes state of the art methods when playing *ASMACAG*. Future work will focus on assessing the performance of *NTBOE* in other multi action games.

References

[1] N. Justesen, T. Mahlmann, S. Risi, J. Togelius, Playing multiaction adversarial games: Online evolutionary planning versus tree search, IEEE Trans. on Games 10 (2018) 281–291.

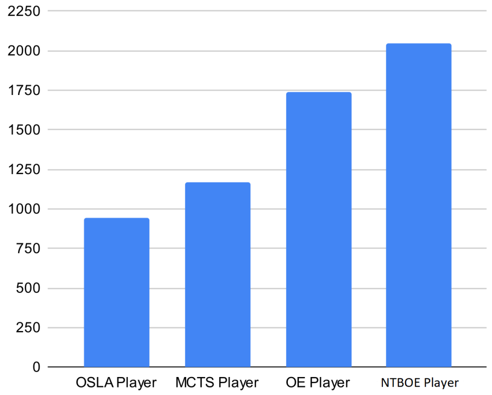

Figure 1: Results chart from second experiment (except games with Random and ties).

[2] S. M. Lucas, J. Liu, D. Perez-Liebana, The n-tuple bandit evolutionary algorithm for game agent optimisation, in: Proc. of IEEE Congress on Evolutionary Computation (CEC'18), 2018.

[3] R. Montoliu, R. D. Gaina, D. Perez-Liebana, D. Delgado, S. M. Lucas, Efficient heuristic policy optimisation for a challenging strategic card game, in: Inter. Conf. on the Applications of Evolutionary Computation (EvoStar'20), 2020.

[4] S. M. Lucas, J. Liu, I. Bravi, R. D. Gaina, J. Woodward, V. Volz, D. Perez-Liebana, Efficient evolutionary methods for game agent optimisation: Model-based is best, in: Game Simulations Workshop (AAAI'19), 2019.

[5] K. Kunanusont, S. M. Lucas, D. Perez-Liebana, Modeling player experience with the n-tuple bandit evolutionary algorithm, in: 14th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment, 2018.

[6] D. Villabrille-Seca, R. Montoliu, Asmacag: Un nuevo juego de cartas multi acción para facilitar el estudio de técnicas de inteligencia artificial, in: 1er Congreso Internacional de DiGRA España 2021 (DIGRAES21), 2021.

[7] N. Justesen, T. Mahlmann, J. Togelius, Online evolution for multi-action adversarial games, in: EvoApplications, 2016.

[8] D. Pérez, S. Samothrakis, S. Lucas, P. Rohlfshagen, Rolling horizon evolution versus tree search for navigation in single-player real-time games, in: Proc. of the 15th Annual Conf. on Genetic and Evolutionary Computation, 2013.

[9] H. Baier, P. I. Cowling, Evolutionary mcts for multi-action adversarial games, 2018 IEEE Conf. on Computational Intelligence and Games (CIG'18) (2018) 1–8.

[10] P. Cowling, E. Powley, D. Whitehouse, Information set monte carlo tree search, IEEE Trans. on Computational Intelligence and AI in Games 4 (2012) 120–143.

[11] R. Myerson, Game Theory: Analysis of Conflict, Harvard University Press, 1997.

[12] R. Coulom, Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search, in: 5th Int. Conf. on Computer and Games, 2006.

[13] G. Chaslot, S. Bakkes, I. Szita, P. Spronck, Monte-carlo tree search: A new framework for game ai, in: Proc. of the 4th AAAI Conf. on AI and Interactive Digital Entertainment, 2008.