Efficient Heuristic Policy Optimisation for a Challenging Strategic Card Game

Raúl Montoliu[1], Raluca D. Gaina[2], Diego Perez-Liebana[2], Daniel Delgado[1], Simon M. Lucas[2]
[1] Institute of New Imaging Technologies.
University Jaume I of Castellón
Castellón, Spain
$\{montoliu, delgadod\}@uji.es$
[2] Game AI Research Group
Queen Mary University of London
London, UK
$\{r.d.gaina, diego.perez, simon.lucas\}@qmul.ac.uk$

Abstract

Turn-based multi-action adversarial games are challenging scenarios in which each player turn consists of a sequence of atomic actions. The order in which an AI agent runs these atomic actions may hugely impact the outcome of the turn. One of the main challenges of game artificial intelligence is to design a heuristic function to help agents to select the optimal turn to play, given a particular state of the game. In this paper, we report results using the recently developed N-Tuple Bandit Evolutionary Algorithm to tune the heuristic function parameters. For evaluation, we measure how the tuned heuristic function affects the performance of the state-of-the-art evolutionary algorithm Online Evolution Planning. The multi-action adversarial strategy card game *Legends of Code and Magic* was used as a testbed. Results indicate that the N-Tuple Bandit Evolutionary Algorithm can effectively tune the heuristic function parameters to improve the performance of the agent.

1 Introduction

In turn-based multi-action adversarial games, each player turn consists of several atomic 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. They need a heuristic function (also known as fitness function) to estimate a score about how good or bad it is to be in a particular state of the game. Heuristic functions tend to have several parameters that should be tuned to obtain good results. Most of the previous works manually tuned the parameters using expert knowledge and experience playing the game (Justesen, Mahlmann, and Togelius 2016), (Baier and Cowling 2018). An alternative solution is to use an evolutionary algorithm to find the best parameter combination.

Recently, the N-Tuple Bandit Evolutionary algorithm (*NTBEA*) (Lucas, Liu, and Perez-Liebana 2018) was presented as an effective method for parameter tuning. It is very useful when the evaluation function of the game is noisy and fairly expensive in CPU time, as is the case in multi-action adversarial games. Hence, it is desirable to have an evolutionary algorithm that can operate very efficiently, making the best possible use of the available fitness evaluation budget, and also one that is robust to noise. *NTBEA* satisfies both criteria. It has successfully been used to tune agent parameters (Lucas et al. 2019). In this paper, *NTBEA* will be used to tune the parameters of the heuristic function. This is the main novelty of the paper.

The performance of the state-of-the-art evolutionary algorithm Online Evolutionary Planning (OEP) (Justesen, Mahlmann, and Togelius 2016), with the heuristic function parameters tuned using *NTBEA*, was assessed. It has been previously tested on the game *HeroAIcademy*. Instead, in this paper, the multi-action adversarial strategy card game Legends of Code and Magic (*LOCM*) (Kowalski and Miernik 2019) was used as testbed. *LOCM* is a game with some similarities to the popular *Hearthstone*[1]. Some of the more challenging features of the game are: 1) a variable number of actions can be played in each turn, 2) some parts of the game are unknown in the state, 3) the order in which the cards are played is very relevant, and 4) the game has a very large branching factor.

One challenging difference of *LOCM* with respect to *Hearthstone* is that, in the former, the time budget to obtain the next turn is just 200 ms in contrast to 60000 ms of a typical *Hearthstone* game. In addition, in the *LOCM* game, the board is divided into two lanes (left and right) instead of just one as in *Hearthstone*. This adds an even higher branching factor since the agents have to deal with the problem of having to decide in which part of the board each card should be played. Figure 1 shows a screenshot of this game.

Summarizing, this paper presents three main novelties:

- We apply the N-Tuple Bandit Evolutionary algorithm to adjust the parameters of the heuristic function used by an evolutionary agent. As far as we know, this is the first paper applying *NTBEA* for this purpose.
- This is the first research paper using the multi-action adversarial game *Legends of Code and Magic* as a testbed. This game has as principal challenging characteristics its very low time budget to obtain player actions in a turn (200ms) and a very large branch factor. In addition, this is the first time that *NTBEA* is used in a game as challenging as *LOCM*.

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1] https://playhearthstone.com

Figure 1: A screenshot of the *Legend of Code and Magic* game during the battle phase (https://www.codingame.com). The cards in the opponent player hand (top) are unknown during the match.

- We assess the performance of the Online Evolution Planning algorithm, with the parameters of their heuristic function tuned using *NTBEA*. As far as we know, this is the first paper applying this type of evolutionary algorithm to a strategy card game.

2 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 in (Cowling, Powley, and Whitehouse 2012). More detail on the game theory concepts can be found in standard textbooks on this subject, e.g. (Myerson 1997).

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 $\rho(s) \in \{0, \ldots, k\}$, 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 $t = 0, 1, 2, \ldots$, 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 though 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 $A(s)$.

The *transition function* Φ maps a (state, action) pair (s_t, a) to a resulting state s_{t+1} by choosing an edge (s_t, s_{t+1}).

A *policy* for player i maps each state s with $\rho(s) = i$ to a probability distribution over $A(s)$. This distribution specifies how likely the player i is to choose each action from that state. One fundamental problem to be solved in adversarial game AI is to find the policy that leads to the highest expected reward, given that all other players are trying to do the same.

The terms *action* and *atomic action* can be confused in multi-action games. In this paper, we will refer to *turn* as every edge outgoing a state (i.e. the actions) and to *atomic action* to every element that can be part of a turn. The set of all possible atomic actions given a state s is denoted $\Gamma(s) = \{\gamma_1, \gamma_2, \ldots, \gamma_{q_s}\}$. The edges outgoing the state s, are all the possible permutations of the elements of $\Gamma(s)$ which, depending on the number of elements, can be a very large number. Therefore, there are as many different turns that can be played from a state s, as possible permutations in $\Gamma(s)$. In most games, the number of atomic actions q_s included in the set $\Gamma(s)$ depends on the state s.

We denote $\tau(s) = [\gamma_1, \gamma_2, \ldots, \gamma_{p_s}]$ as the list of ordered atomic actions, included in $\Gamma(s)$, that are played for player $\rho(s)$. Note that each atomic action can be played just once in a turn and that not all atomic actions included in $\Gamma(s)$ must be played, i.e. $p_s <= q_s, \forall s$. The number of elements included in the list $\tau(s)$ can be different depending on the game and the policy used.

The problem that the agents have to solve is, given a state s, to find the list $\tau(s)$ that leads to the highest immediate expected reward, i.e. to find the best subset of atomic actions allowed from the state s and the appropriate order to play them.

The heuristic function $\lambda(s)$ is frequently defined as a linear combination of some features included in the state as follows:

$$\lambda(s) = \sum_{f=1}^{F} \omega_f * \theta_f \tag{1}$$

where F is the number of features of the state being taken into account when estimating the reward vector. Therefore, the problem to be solved in this paper is to find the correct value of each weight ω_f. In this paper, we proposed to use *NTBEA* for this purpose.

In the *LOCM* game, there are only two players ($k = 2$) and there is no *environment player*. The number of atomic actions q_s included in $\Gamma(s)$ can vary along with the state. Preliminary experiments have shown that this number can vary from 1 to 35, with two peaks around 5 and 15 actions as most common cases. These statistics have been calculated after running 100 matches where the *OEP* agent (explained in detail in Section 3.3) has played versus itself. Similar results can be obtained when using other agents. In addition,

there is no fixed number of atomic actions q_s.

3 Related Work and Background

3.1 Related work on strategy card games

Most of the previous works on Strategy Card Game AI deal with the *Hearthstone* game. Since the cards in the opponent's deck should be unknown in this type of games, some authors have developed methods to predict them. Dockhorn et al. (Dockhorn et al. 2018) used the knowledge gathered from a database of human replays to create a knowledge-base of frequent player card combinations. Bursztein presented in (Bursztein 2016) a similar work where a bag-of-words of card co-occurrence bi-gram was used for training a prediction system for the next upcoming card.

Santos et al. (Santos, Santos, and Melo 2017) proposed a modified version of the Monte Carlo Tree Search (*MCTS*) algorithm which integrates expert knowledge in the algorithms search process through a database of decks that the algorithm uses to cope with the imperfect information and through the inclusion of a heuristic that guides the *MCTS* rollout phase. Zhang and Buro (Zhang and Buro 2017) improved the effectiveness of MCTS by reducing search complexity in the selection phase and by improving the rollout phase, in which *MCTS* will sample action sequences according to a rollout policy. Choe and Kim. (Choe and Kim 2019) presented an MCTS-based approach to reduce the complexity of the search space and decide on the best strategy. They used state abstraction to present the search space as a Directed Acyclic Graph (DAG) and introduced a variant of Upper Confidence Bound for Trees algorithm for the DAG. In addition, they applied a sparse sampling algorithm to handle imperfect information and randomness and reduce the stochastic branching factor.

3.2 N-Tuple bandit evolutionary algorithm

The N-Tuple Bandit Evolutionary algorithm (*NTBEA*) (Lucas, Liu, and Perez-Liebana 2018) combines multi-armed bandits with an evolutionary algorithm to provide a sample-efficient optimization algorithm. *NTBEA* was developed to handle noisy optimization problems in a sample-efficient way. The algorithm analyses the contribution of each individual parameter value, together with combinations of parameter values. Each combination is referred to as an N-Tuple. A modified form of the bandit equation is used to balance exploiting apparently good parameter settings with exploring those that have not yet been sampled much. *NTBEA* assumes that the execution time of querying the bandit landscape model is negligible compared to evaluating a candidate solution on the target problem.

The algorithm starts sampling a single solution point uniformly at random in the search space. The fitness of this solution is evaluated once, in the problem domain, using the noisy evaluator. This solution is then stored in the bandit fitness landscape model, together with its fitness value. The model is then searched within the neighbourhood for a new solution. The neighbourhood is defined using the number of neighbours and the proximity distribution to the current solution, which is controlled by a mutation operator. The neighbouring solution with the highest estimated Upper Confidence Bounds value (UCB) is then selected and the process repeats until the evaluation budget is used up, or some other termination condition is met.

For a problem with F parameters to be tuned, *NTBEA* is typically configured using F 1-Tuple bandits, $\frac{F*(F-1)}{2}$ 2-Tuples bandits and one F-Tuple macro-arm bandit.

A key part of the algorithm is the value function used to sample in a large search space. The UCB value of any arm i is defined as:

$$UCB_a = X_a + C * \sqrt{\frac{\log n}{n_a + \epsilon}} \tag{2}$$

where X_a is the mean reward for playing arm a. This is the exploitation term. The right-hand term controls the exploration, where n is the total number of times this bandit has been played, and n_a is the number of times the arm a has been played. The term C is called the exploration factor: higher values of C lead to exploratory search, low values lead to a more greedy or exploitative search. The ϵ value is used to control whether each arm should be pulled at least once. In the standard UCB formula, ϵ is set to zero ensuring that each arm is pulled once in turn, but for *NTBEA* purposes, this would be impractical, as the macro-arm consisting of the entire N-Tuple would force an exhaustive exploration of the search space.

The UCB aggregate value for all arms used, given a solution x is defined as follows:

$$UCB(x) = \frac{\sum_{j=1}^{m} UCB_{N_j(x)}}{m} \tag{3}$$

where m is the total number of bandits used and N is the N-Tuple indexing function such that $N_j(x)$ indexes the $j-th$ bandit for search space point x.

3.3 Online evolutionary planning

Online Evolutionary Planning (*OEP*) (Justesen, Mahlmann, and Togelius 2016) is an evolutionary approach that can be applied to multi-action adversarial games. It optimises the action sequence of the current turn, without looking ahead to future turns of the player or the opponent. *OEP* begins its search by creating an initial population of genomes. Each genome represents a complete turn, a fixed-length sequence of atomic actions. These atomic actions can be sampled randomly, or another initialisation strategy can be used. The population is then improved from generation to generation until computation time runs out or the allowed budget is depleted.

The crossover step of the evolutionary algorithm used can lead to an illegal action in some of the created offspring. A repair strategy is needed, for instance by picking random legal actions or using a greedy approach (Baier and Cowling 2018). A proportion of the offspring undergoes mutation. One randomly chosen action of the sequence is changed to another action randomly chosen from all legal actions. If this leads to illegal actions later in the sequence, they are replaced using the repair strategy as well.

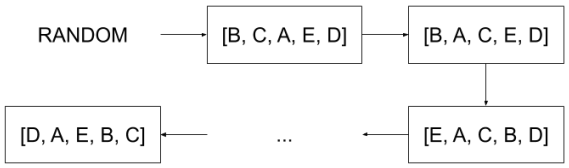

Figure 2: An example of the *OEP* algorithm in a turn with 5 possible atomic actions: A, B, C, D and E. In this example, a 1+1 evolutionary algorithm is used.

When the time budget is exhausted, *OEP* returns the action sequence represented by the current best genome, so it can be executed, one atomic action at a time. It can, therefore, be seen as doing one iteration of rolling horizon evolutionary algorithm (RHEA) (Perez Liebana et al. 2013) at the beginning of each turn, and with a search horizon of one turn. Figure 2 shows an example of the OEP operation where $\Gamma(S) = \{A, B, C, D, E\}$.

4 Methods

4.1 Legend of code and magic

Legends of Code and Magic (*LOCM*) (Kowalski and Miernik 2019) is an implementation of a multi-action adversarial Strategy Card Game, designed to perform AI research. Its advantage over real card game AI engines is that it is much simpler to handle by the agents, and thus allows testing more sophisticated algorithms and quickly implementing theoretical ideas. The most similar commercial game is *Hearthstone*. One important difference is that the time budget in *LOCM* is just 200 ms in contrast to 60000 ms used in *Hearthstone*. Therefore, some algorithms frequently used in *Hearthstone*, such as *MCTS*, do not perform as well here.

LOCM is based on the fair arena mode, i.e., before every game, both players create their decks secretly from symmetrical yet limited choices. Because of that, the deck building phase cannot be simply reduced to using human-created top-meta decks as in *Hearthstone*. All cards effects are deterministic. The non-determinism is introduced by the ordering of cards in the decks.

The game has two phases. First, in the draft phase, for 30 turns, both players are given a choice between 3 different cards. Players select the card they want to add to their deck. Both players can select the same card. Players do not know which cards have been selected by the opponent. Once the draft phase is over, both decks (consisting of 30 cards each) are shuffled. The second phase is the battle, where each player plays cards from their hand on the two lanes, on their side of the board. Each player starts with 30 health points (HP). To reduce the health points of an opponent, the player must make use of cards to deal damage. The game is over once any player reaches 0 or less HP.

There are two different types of cards: Creatures and Items. Placing a Creature (similar to minions in *Hearthstone*) on the board is called summoning. A player summons Creatures to their side of the board by paying their cost in *mana*. They are used to attack the opponent and also serve as a defence against the Creatures of the opposing player. When a Creature attacks the opponent directly (and not their cards), it reduces the HP of the opponent according to its attack strength. When a Creature attacks another Creature, they both deal damage equal to their attack strength to the defence of the other creature. Creatures are removed from play when their defence reaches 0 or less. Creatures can have different abilities. One of the most important ones is *Guard*, which enforces that enemy creatures from the same lane must attack creatures with *Guard* first.

The other type of card is Items (similar to spells in *Hearthstone*). When played, Items have an immediate and permanent effect on the board or the players. There are three types of Items: 1) green Items must target the active player's creatures and hurt them. 2) red Items must target the opponent's creatures and have a negative effect on them. 3) blue Items can be played to give the active player a positive effect or cause damage to the opponent, depending on the card.

Mana is necessary to play cards from the hand. Each turn, the number of mana units that each player can use increases. Each player can spend as much mana units per turn as they have. For instance, if the player starts the turn with 6 mana units, it is possible to summon two cards with cost 3 each, or summon only one with 6 mana cost. The cost of the cards varies from 0 to 12.

To run a turn the agent must provide to the environment a list of atomic actions. The complete rules of the game can be found in (Kowalski and Miernik 2019).

Valid and possible atomic actions As shown in Figure 1, the valid atomic actions that can be played in the actual state of the game are as follows (yellow numbers in the top-right corner of each card show their ID): {*Summon 1 L; Summon 1 R; Summon 3 L; Summon 3 R; Use 2 6; Use 2 7; Use 4 6; Use 4 7; Use 5 8; Use 5 9; Attack 6 9, Attack 7 -1*}. Note that card 6 cannot attack either to opponent card 8 or directly to the opponent since opponent card 9 has the *Guard* property.

But, if the agent decides to play first the atomic action *Use 5 9*, then two new atomic actions can be played for card 6 since now the opponent card 9 will no longer be on the board and no other card with the *Guard* ability remains on the board. These two new possible atomic actions are: *Attack 6 8* and *Attack 6 -1*.

To give to the agents the opportunity to discover this kind of interesting behaviours, all the possible atomic action that can be played (taking into account any possible order of playing the atomic actions) and not only the valid ones, should be included in the set $\Gamma[s]$.

4.2 Agents

All the agents tested in this paper, start from the list of all possible actions that can be played in a state s (i.e. the list $\Gamma(s)$) and attempt to find an optimal turn $\tau(s)$. In all cases, before playing an atomic action, the agent checks if the atomic action is a valid one. If not, this atomic action is ignored and the agent tries to play the next one in the list $\tau(s)$.

Random agent The random agent just shuffles the initial list of atomic actions $\Gamma(s)$. They are then played in the resulting order. The heuristic function is not used.

NoAI agent This agent directly plays the list of initial atomic actions in the original order. This order follows a simple strategy which aims to directly attack the opponent whenever possible. Similarly to the *Random agent*, *NoAI* does not use the heuristic function.

OEP-based agent The original version of the *OEP* algorithm has to select 5 atomic actions (in the game *HeroAIcademy* $q_s = 5, \forall s$) to be played from the set of all valid actions that can be played. Therefore, the genome is a list of five atomic actions. In our case, the genome is the list of all possible (not only valid) atomic actions (i.e. $\Gamma(s)$) that can be played given the state of the game. The agent plays the atomic actions following the order in the list $\Gamma(s)$. If an action is not valid, given the current state of the game, it is ignored and play continues with the next one.

Mutation consists of changing the order of the atomic actions on the list. This allow to discover interesting sequence of actions that can maximise the immediate rewards.

Since the time budget of *LOCM* is very small (200ms), a simple 1+1 evolutionary algorithm has been used. Therefore, no crossover step is needed. In cases where the time budget is higher, a more sophisticated evolutionary algorithm could be used.

4.3 Heuristic function design

The heuristic function $\lambda(s)$ has been designed as a linear combination of several features, balanced by a set of weights (see Equation 1). Five different features are proposed:

- θ_1: The objective of the game is to reduce the health points of the opponent to zero. This feature is the difference between the opponent health points after playing the turn (i.e. at state s_{t+1}), and their initial health points (i.e. at state s_t). A larger value means that the agent is reducing more health points from the opponent.
- θ_2: Agents have to also pay attention to the opponent cards on the board, since, in the next turn, they can be used to attack the player. Therefore, it is very important to balance the attacks between the opponent cards and the opponent directly. This feature is the possible loss of health points if the opponent decides to directly attack the player using all its cards. In this case, small values are preferable.
- θ_3: This feature is used to help the agent to decide which card should be played, as not all the cards are equally useful in each turn. The feature sums the values of all cards on the player's board. The better the cards on the board, the higher the value.
- θ_4: Sometimes it is better not to play some items, in order to allow them to be used in the future. This feature is the sum of the values of all Item cards in the player's hand. The better Item cards in the hand, the higher the value.
- θ_5: Some cards increase the number of cards that are going to be drawn in the next player turn. This feature is the number of cards that will be drawn in the next player turn.

In this paper, we propose to use *NTBEA* to adjust the weights $[\omega_1, \ldots, \omega_5]$. The noisy evaluator needed for *NTBEA* consists of playing several games between the *OEP* agent using the tuned heuristic function and the *NoAI* agent. The fitness of a set of weights is determined by the win rate of the *OEP* agent in the evaluation games played. Several plays are needed since the results of a game between two agents strongly depend on the decks used.

4.4 The deck problem

The outcome of the agents strongly depends on the deck used and on the order of the cards. To deal with this problem, and therefore to allow a fair comparison of the agents, several decks have been built in advance.

The draft step of the game has been run 10 times. Each time, 3 different strategies have been used to select one of the 3 cards proposed by the system. The first strategy (employed by the winner in the CEC19 *LOCM* competition) selects the card with the best score based on a hand-designed estimation of the value of each card. The second strategy (employed by the runner up in the CEC19 competition) tries to obtain a certain number of cards with low mana cost (useful for the beginning of the game), and a certain amount of moderate and high cost (useful for the middle and last turns of the game). The third strategy is a combination of both ideas: when the system provides 3 cards falling in the same mana cost group, the best according to the first strategy is selected. Once the 3 decks have been obtained, they are shuffled 10 times, producing 10 different orders to play each deck.

In total, a database of $3*10*10 = 300$ different decks was obtained. The battle phase of the game was then modified to replace the draft phase with direct use of the pre-built decks.

5 Experiments and Results

5.1 Experimental set up

In this paper, we use terminology adopted from tennis to compare the performance of the agents. A *game* is a play between two agents using a particular deck and a particular order. The result of a game is 1 if the first player is the winner or 0 if the second player wins.

A *set* is a collection of 12 games between two agents. In a set, 3 decks $\mathcal{D}_1^{d,o}$, $\mathcal{D}_2^{d,o}$ and $\mathcal{D}_3^{d,o}$ are used (where d and o stand for the $d-th$ draft used to generate the 3 decks and the $o-th$ order) to produce a fair comparison between agents, since all possible combination of games using the 3 decks are taken into account. Additionally, each agent plays half the games as first player, and half as the second player.

Finally, a *match* is to play several sets varying the draft (d) from where the 3 decks and their order (o) are selected. In total, a match consists of 100 sets, or 1200 games between two agents. The result of a match is the mean win rate per deck for the first agent.

NTBEA was used to tune the weights of the heuristic function (i.e. $[\omega_1, \ldots, \omega_5]$). For all features, less ϕ_2, six different values were tested: $[0, 1, 2, 3, 4, 5]$. In the case of the feature ϕ_2 the following values were used instead: $[-5, -4, -3, -2, -1, 0]$. 100 neighbours and a mutation probability of 10% were used.

For the noisy evaluator, a set was played using the *OEP* agent, with the solution being the current one in *NTBEA*, versus the *NoAI* agent. Each time the noisy evaluator was used,

Table 1: Win rate mean and standard deviation (for all decks) of matches between two agents. μ and σ stand for mean and standard deviation, respectively.

1st agent	2nd Agent	μ	σ
OEP([1,1,0,0,0])	*OEP([1,0,0,0,0])*	0.68	0.12
OEP([1,0,1,0,0])	*OEP([1,0,0,0,0])*	0.81	0.11
OEP([1,0,0,1,0])	*OEP([1,0,0,0,0])*	0.55	0.12
OEP([1,0,0,0,1])	*OEP([1,0,0,0,0])*	0.50	0.11

d and o were randomly obtained, so as to prevent overfitting the result to a particular deck. Note that a brute force algorithm needs to play $6^5 * 1200 \approx 0.9 * 10^7$ games to explore the complete search space. The use of *NTBEA* can drastically reduce the number of times that a game is played.

The number of calls to the heuristic function was used as a time budget for the agents, therefore, the agents performance is independent of the speed of the computer used to run the experiments. All the experiments reported in this paper limit the agent budget to 500 calls.

As an average, the *OEP* agent needed 54.98 ms (with $\sigma = 19.74$) to complete the budget. The experiments were run in a Dell XPS 15 9570 computer with an Intel(R) Core(TM) I9-8590HK CPU @ 2.9GHz processor and 32MB of RAM.

5.2 Preliminary experiments

The main objective of the game is to reduce the health points of the opponent to 0. As a first experiment, we conducted a match between the agents *OEP(1s)* and *OEP([1,0,0,0,0])*. The first is an agent with the same weights for all features (i.e. $\omega_f = 1, \forall f$). The second agent has all weights set to zero less the first one, i.e. an agent that only take into account the reduction of opponent health points (feature θ_1).

The results obtained show the first agent winning 85.6% of the games. Thus we can conclude that only taking into account θ_1 does not produce the best results and a combination of all features is suggested to work better.

Table 1 shows the results of several matches where the first agent is *OEP* taking into account various combinations of two features, versus the *OEP(1s)* agent. This aims to study how important the features $\theta_2, \theta_3, \theta_4, \theta_5$ are.

The results obtained suggest that feature θ_3 (related to the value of the cards on the board) has a strong relevance. Furthermore, θ_2 (related to the opponent damage threat), seems to also be important. However, features θ_4 and θ_5 appear to have less influence.

5.3 *NTBEA* evolution and recommend solution

Figure 3 shows the evolution of the parameters tuned by *NTBEA*. Every 100 iterations the recommended solution given the actual state of the *NTBEA* process has been estimated. At the beginning of the process, the parameters vary a lot, since *NTBEA* had not had enough time to find a good solution in a so noisy scenario as *LOCM*. Around 5000 iterations, the algorithm converges for all parameters.

The final solution recommended by *NTBEA* is $[3, 2, 5, 1, 1]$. In the recommended solution, all features are considered important (none were set to 0) with $\theta 3$ and θ_1 being given most importance.

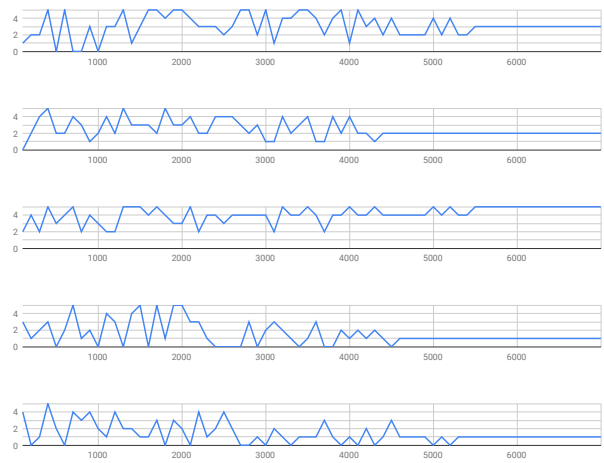

Figure 3: Evolution of the parameters when using *NTBEA*. From top to bottom, the figures show the evolution of the 5 parameters (starting with θ_1). The x-axis is the number of iterations. The y-axis shows the parameter value.

Table 2: Win rate mean and standard deviation (for all decks) of matches between two agents. μ and σ stand for mean and standard deviation, respectively.

1st agent	2nd Agent	μ	σ
NoAI	*Random*	0.767	0.057
OEP(1s)	*Random*	0.901	0.039
OEP(NTBEA)	*Random*	0.933	0.036
OEP(1s)	*NoAI*	0.803	0.057
OEP(NTBEA)	*NoAI*	0.825	0.058
OEP(1s)	*EMCTS*	0.505	0.022
OEP(NTBEA)	*EMCTS*	0.560	0.021

According to the results obtained, it is not only important to reduce the health of the opponent (θ_1) as the *NoAI* agent tries to do. To have good cards on the board is the most important feature (θ_3) and it is also relevant to take into account the amount of damage that the opponent can produce in the next turn (θ_2). The solution obtained using *NTBEA* agrees with the results obtained in Section 5.2.

5.4 *OEP* Agent performance

In this section, several matches have been tested to study the performance of the *OEP* agent when using the heuristic function tuned by *NTBEA* (we call this agent *OEP(NTBEA)*) versus some baselines.

We first test the performance of the *NoAI*, *OEP(1s)* and *OEP(NTBEA)* agents versus *Random*. Figure 4 shows the result obtained for each deck. First three rows of Table 2 show the mean (and standard deviation) across all decks.

Results shows that *NoAI* can win a lot of games against *Random* (76.7%) since it uses a sensible strategy. Using all the cards on the board to directly attack the opponent is a successfully and aggressive strategy that requires the op-

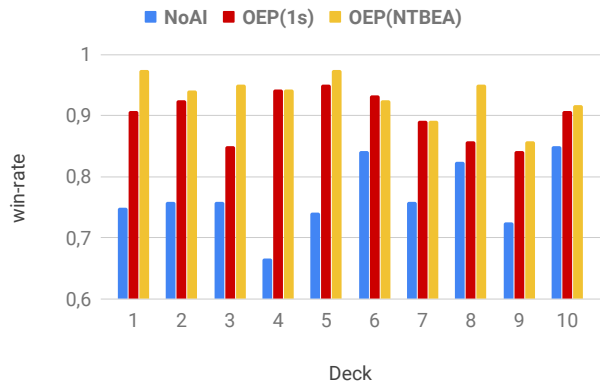

Figure 4: Mean win rate for each deck for matches played by *NoAI*, *OEP(1s)* and *OEP(NTBEA)* versus Random.

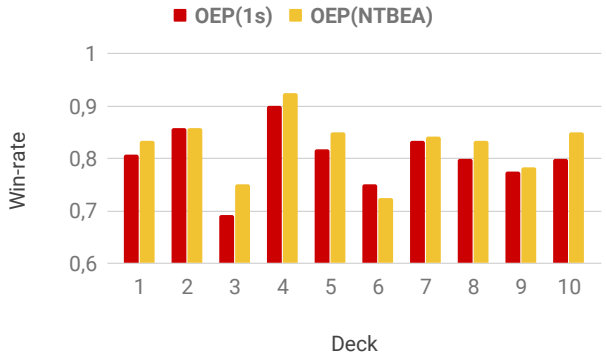

Figure 5: Mean win rate for each deck for matches played by *OEP(1s)* and *OEP(NTBEA)* against *NoAI*.

ponent player to develop an appropriate defence. Results also show that *OEP(1s)* clearly outperforms *NoAI* winning the 90% of the games. Finally, *OEP(NTBEA)* obtains even better results than *OEP(1s)*, suggesting that the combination found by *NTBEA* improves the performance of *OEP*. *OEP(NTBEA)* has better win rate for all decks less one.

For a second experiment, we test *OEP(1s)* and *OEP(NTBEA)* agents versus *NoAI*. Figure 5 and the fourth and fifth rows of Table 2 show the results of those matches. *OEP(NTBEA)* outperforms the original algorithm with almost all decks, with a better mean win rate.

The last experiment aims to study whether the tuned heuristic is also useful when playing against a different agent to the one used during training. For this purpose, both *OEP(1s)* and *OEP(NTBEA)* agents played a match against an agent that uses the Evolutionary Monte Carlo Tree Search algorithm (EMCTS) (Baier and Cowling 2018).

EMCTS combines some of the ideas of tree search from *MCTS* with the genome-based approach of evolutionary algorithms. *EMCTS* starts from a complete sequence of atomic actions, just like the genomes of *OEP*. *EMCTS* grows a tree by mutating the current sequence, using the same mutation

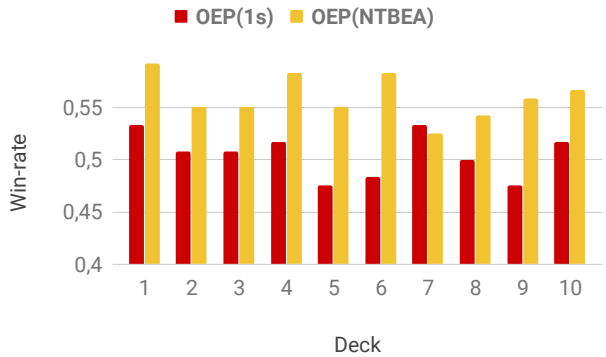

Figure 6: Mean win rate for each deck for matches played by *OEP(1s)* and *(NTBEA)* against *EMCTS*.

operator as *OEP*. *EMCTS* does not use rollouts to complete the game, it simply evaluates the sequences at the leaf nodes. The backpropagation step is unchanged from MCTS. When the time budget is exhausted, the tree is traversed until a node without descendants is found. The best sequence of atomic actions in the path is the one returned by the algorithm, to be played by the agent. We have used the same heuristic function design as in the *OEP* algorithm and all the weights have been set to 1.

Figure 6 and last two rows of Table 2 show the results obtained. They show that the training of *OEP(NTBEA)* was robust enough to also obtain better results for almost all decks against the different opponent. As Figure 6 and Table 2 show, *OEP(1s)* has a very similar performance to *EMCTS*, winning close to 50% of matches, i.e. none of the two agents clearly outperforms the other one. However, when using the weights tuned by *NTBEA*, the win rate increases such that the *OEP*-based agent is preferable in almost all decks.

6 Conclusions

This paper presented the use of the N-Tuple Bandit Evolutionary Algorithm (*NTBEA*) to optimise the parameters of the heuristic function. The challenging strategic card game *Legends of Code and Magic* was used to test the proposed approach. The results obtained demonstrate that *NTBEA* can effectively tune the heuristic function weights to improve the performance of the evolutionary algorithm used, Online Evolutionary Planning (*OEP*), even when playing against opponents different to those used during training.

Future work must be focused on studying whether the parameters obtained with *NTBEA* can be applied when other decks are used. In addition, some other N-tuple configurations could also be studied.

Acknowledgements

The authors want to thank J. Kowalski for his help solving doubts about the *LOCM* rules and N. Justensen and H. Baier for their help in better understanding their algorithms.

References

Baier, H., and Cowling, P. I. 2018. Evolutionary mcts for multi-action adversarial games. In *Preceedings of the 2018 IEEE Conference on Computational Intelligence and Games (CIG'18)*.

Bursztein, E. 2016. I am a legend: Hacking hearthstone using statistical learning methods. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*.

Choe, J. S. B., and Kim, J.-K. 2019. Enhancing monte carlo tree search for playing hearthstone. In *Proceedings of the 1st Conference on Games (GOG'19)*.

Cowling, P.; Powley, E.; and Whitehouse, D. 2012. Information set monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games* 4:120–143.

Dockhorn, A.; Frick, M.; Akkaya, Ü.; and Kruse, R. 2018. Predicting opponent moves for improving hearthstone AI. In *Information Processing and Management of Uncertainty in Knowledge-Based Systems. Theory and Foundations*, volume 854, 621–632. Springer.

Justesen, N.; Mahlmann, T.; and Togelius, J. 2016. Online evolution for multi-action adversarial games. In *Proceedings of the 19th European Conference on Applications of Evolutionary Computation (EvoApplications'16)*, volume 9597 of *Lecture Notes in Computer Science*, 590–603. Springer Verlag.

Kowalski, J., and Miernik, R. 2019. Legends of code and magic. https://jakubkowalski.tech/Projects/LOCM/. [Online; accessed 26-May-2019].

Lucas, S. M.; Liu, J.; Bravi, I.; Gaina, R. D.; Woodward, J.; Volz, V.; and Perez-Liebana, D. 2019. Efficient evolutionary methods for game agent optimisation: Model-based is best. In *AAAI Workshop on Games and Simulations for Artificial Intelligence*.

Lucas, S. M.; Liu, J.; and Perez-Liebana, D. 2018. The n-tuple bandit evolutionary algorithm for game agent optimisation. In *Proceedings of IEEE Congress on Evolutionary Computation (CEC'18)*.

Myerson, R. 1997. *Game Theory: Analysis of Conflict*. Harvard University Press.

Perez Liebana, D.; Samothrakis, S.; Lucas, S.; and Rohlfshagen, P. 2013. Rolling horizon evolution versus tree search for navigation in single-player real-time games. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation (GECCO'13)*, 351–358.

Santos, A.; Santos, P. A.; and Melo, F. S. 2017. Monte carlo tree search experiments in hearthstone. *2017 IEEE Conference on Computational Intelligence and Games (CIG'17)* 272–279.

Zhang, S., and Buro, M. 2017. Improving hearthstone AI by learning high-level rollout policies nd bucketing chance node events. In *2017 IEEE Conference on Computational Intelligence and Games (CIG'17)*, 309–316.