Evolving Systems (2021) 12:169–176
https://doi.org/10.1007/s12530-020-09351-4

ORIGINAL PAPER

A novel real-time design for fighting game AI

Gia Thuan Lam[1] · Doina Logofătu[2] · Costin Bădică[3]

Received: 7 November 2019 / Accepted: 5 July 2020 / Published online: 18 July 2020

Abstract
Real-time fighting games are challenging for computer agents in that actions must be decided within a relatively short cycle of time, usually in milliseconds or less. That is only achievable by either very powerful machines or state-of-the-art algorithms. The former is usually a costly option while the latter remains an ongoing research topic despite countless research. This paper describes our algorithmic approach towards real-time fighting games via the fighting game AI challenge. The focus of our research is the LUD division, the most challenging category of the competition where action data is hidden to prevent methods that are dependent on prior training. In this paper, we propose several generic heuristics that can be used in combination with Monte-Carlo tree search. Our experimental results show that such an approach would provide an excellent AI outperforming pure Monte-Carlo tree search and classic algorithms such as evolutionary algorithms or deep reinforcement learning. Nonetheless, we believe that our proposed heuristics should be able to generalize to other domains beyond the scope of the fighting game AI challenge.

Keywords Real-time fighting games · Fighting game AI challenge · Generic heuristics · Monte-Carlo tree search

1 Introduction

The evolution of artificial intelligence (AI) over the past decades has given rise to human-defeating computer programs. In various areas, computer programs no longer aim for human intelligence, but they have already surpassed their human counterparts. Probably the best-known example is the area of chess with multiple historic victories against human experts such as Deep Blue (Campbell et al. 2002), AlphaGo (Silver et al. 2016), and AlphaGo Zero (Silver et al. 2017). The success of these AI programs is exciting for the fact that their performance is achieved by self-learning with minimal intervention from humans. Such a feat would not have been possible without numerous advancements in computer hardware and the emergence of multiple learning paradigms such as evolutionary algorithms (EA) (Back 1996; Balabanov and Logofătu 2019), deep learning (DL) (Nork et al. 2018; Skinner et al. 2019), and Monte-Carlo tree search (MCTS) (Chaslot et al. 2008; Logofatu et al. 2019). That raises the question for many AI researchers if human guidance is already irrelevant for AI in this day and age.

Fortunately, such successes are not ubiquitous but were partly thanks to the nature of their task. Playing chess is a turn-based environment that gives modern supercomputers sufficient time to process. However, under real-time circumstances where programs must respond reasonably well within a short instance of time, an average home processor proves insufficient against the infinite number of game states. In such conditions, most popular learning algorithms would suffer from a severe loss in performance and become more dependent on human analysis for improvement.

Fighting game (Harper 2013) is one such challenge where 2 or more characters competing against each other until only one remains. This is a classic category of games favored by countless children around the world. Some popular games of this category can be listed such as Street Fighter (Ng 2006), Mortal Kombat, Fatal Fury, Art of Fighting, The King of

✉ Doina Logofătu
logofatu@fb2.fra-uas.de

Gia Thuan Lam
cs2014_thuan.lg@student.vgu.edu.vn

Costin Bădică
cbadica@software.ucv.ro

[1] Faculty of Engineering, Vietnamese-German University, Thu Dau Mot, Vietnam

[2] Department of Computer Science and Engineering, Frankfurt University of Applied Sciences, Frankfurt, Germany

[3] Department of Computer Sciences and Information Technology, University of Craiova, Craiova, Romania

Fighter, and much more. They can vary from the number of players to character design, but one among their common properties is the instantaneous interaction between players. As no player is likely to wait for the others to perfect their strategy, the decision of an appropriate attack must be issued in a timely manner. In the language of computer science, it is a real-time requirement.

The fighting game AI challenge (Lu et al. 2013) is an AI competition that provides a platform for implementing AI fighters. The challenge's target is to solve the general real-time fighting game. In other words, AI programs must operate under real-time constraints with limited home computing resources. LUD is one of the 3 competition categories in this challenge. By possessing characteristics that prevent the use of training-dependent methods, it has an extra layer of complexity. Through our participation in this challenge and especially the LUD division, we propose an algorithm design that can reduce the negative influence of our limited computational resources. Our solution is a proper blend between a generic case-analysis approach with the winning MCTS algorithm. The result shows that by blending some human wisdom with MCTS, the resulting performance is superior to all in existence.

2 The fighting game AI challenge

This AI challenge is initiated by the Intelligent Computer Entertainment Lab from Ritsumeikan University and is hosted annually by a prestigious international conference in games and AI. For instance, it was hosted by the IEEE Conference on Computational Intelligence and Games (CIG) between 2014 and 2018. This section outlines the challenge's technical problems and the category of our focus–LUD division.

2.1 Game overview

The game consists of 2 characters fighting against each other using actions resembling martial arts. A player will be considered *dead* once his *health points* (HP) decreases to 0. In each cycle of time (called *frame*, approximately 16.67 ms), each player will input a game state and must issue an appropriate action depending on the given state and his remaining *energy*. The victory goes to one who remains alive (and the other is dead) after 3600 frames. Otherwise, the result will be considered a draw. Figure 1 illustrates a sample fighting game.

2.2 Action parameters

The issuing of an action $a = (p_1, p_2, \ldots, p_n)$ for the current game state is based on a variety of deciding factors p_i such as:

- p_1: How much damage can a inflict on the opponent?
- p_2: Do we have sufficient energy to perform a?
- p_3: Is a fast enough so that the opponent cannot escape?
- p_4: Is the opponent within the range of a?
- ...
- p_n: How much energy will remain after performing a?

The set of all such factors defines the action a and are indispensable in the decision-making process for a given game state.

2.3 LUD division and technical challenges

If the set of actions $\mathcal{A} = (a_1, a_2, \ldots, a_n)$ is predefined, for a game state $\mathcal{S}$, it is possible to identify the next state

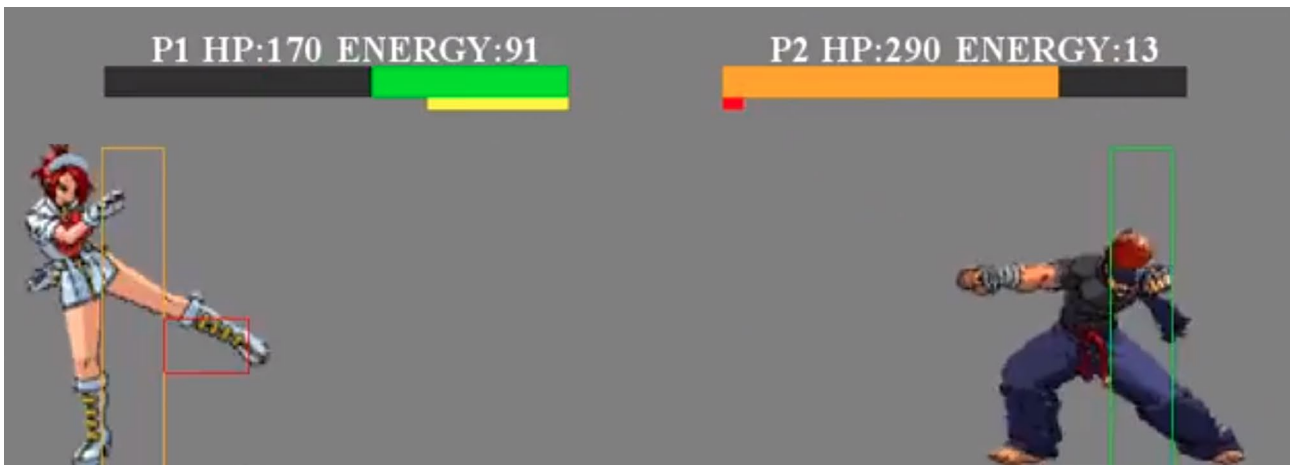

Fig. 1 A sample fighting game

$\mathcal{S}\simeq = \phi(\mathcal{S}, a)$ where ϕ stands for the transition function and $a \in \mathcal{A}$. In this scenario, we can do multiple game simulations and generate the data which can be fed into numerous data-based learning algorithms such as reinforcement learning or even case-analysis. This approach will result in a trained model that can be deployed during the game such as a parameterized artificial neural network (ANN) or a memorized table for simple if-then-else approaches. Such a method is feasible given that we have sufficient preparation time before the battle to generate data and train our models as much as possible. Nonetheless, what would happen if ϕ is undefined?

In LUD division, the action data will only become available during the game, which is why ϕ cannot be determined before that. Without ϕ, it is impossible to simulate games and generate battle data in advance, making data-based learning methods unusable. To make it worse, real-time constraints make data generation and model training during the short duration of the game unrealistic.

To conclude, the technical difficulty of the LUD division derives from the hiding of action data and real-time constraints. Compared to traditional AI problems such as playing chess, those conditions create extra complexity by eliminating data-based learning methods. That enables the reign of simulation-based methods such as MCTS given their advantage that no prior training is necessary.

3 Related work

Fighting game AI is a very classic area of research with various contributions (Thawonmas and Osaka 2006; Osaka et al. 2014; Cavazza 2000; Tamassia 2017). Previous work was mostly based on traditional memorization methods: training a machine learning model from simulated battle data. Data can be generated from some similar environments where action parameters are available. The resulting trained model is then deployed for competing in LUD-like environment in which action data is hidden. Consequently, since the data is just similar but not the same, such approach could not provide satisfactory performance. Typical methods of this type include reinforcement learning (Pinto and Coutinho 2018; Taylor et al. 2014), deep Q networks (Yoon et al. 2017), Artificial Neural Networks (Robison 2017; Cho et al. 2006), K-nearest neighbors (Nakagawa et al. 2014, 2015), or just memorized case-analysis tables Kim and Kim (2017).

Previous competition results of the fighting game AI challenge show that the most successful solutions belonged to those who could lessen the dependence on training data. In other words, algorithms that do not rely on prior training would work best in this scenario. The most successful methods of this category are Monte-Carlo tree search (MCTS) (Yoshida et al. 2016; Ishihara et al. 2016; Ishii et al. 2018) and Evolutionary Algorithms (Kim et al. 2018; Martinez-Arellano et al. 2016; Kim et al. 2019). Part of the reason behind their superior performance compared to data-based learning methods is because they are unaffected by the bias of training data. However, even these methods are not without their problems due to real-time requirements. A simulation-based method like MCTS requires numerous game simulations and population-based techniques such as evolutionary algorithms need to evaluate the entire population. Given the short duration of a cycle (16.67 ms), they pose a serious performance problem.

Last, but not least, a lesser-known approach that has proven very effective in the history of this challenge is to discover hidden holes inside the game. They are programming bugs and not restricted only to fighting games, but the player knowing them would be given overwhelming advantages compared to the other. Nevertheless, such holes are rare and hard to find. A relevant publication can be found in Zuin et al. (2016).

Most contributions in this area concentrated on a single technique. For instance, the work of Yoshida et al. (2016) is one such approach that is 100% only about MCTS. Our work differs by being not a single algorithm but an algorithm design that incorporates all mentioned approaches. By generalizing a case-analysis to lessen its dependence on actual data and exploiting a hidden hole in the game, we cover most scenarios and only resort to MCTS whenever necessary. Furthermore, compared against our previous contribution Thuan et al. (2019), this paper presents more generic ideas that can be generalized beyond the scope of the Fighting Game AI challenge. This ultimate combination preserves the effectiveness of MCTS while the overall performance is significantly improved. Table 2 in Sect. 5 is the most convincing example to showcase our improvements. The AI program named *SampleMctsAi*, for example, was implemented 100% based on the paper by Yoshida (100% pure MCTS). As can be seen, the results demonstrated our superior performance in all categories.

4 Contribution

As mentioned above, this paper contributes an algorithm design that combines a generic case-analysis and MCTS. By so doing, the resulting program would handle most of the game states with simple case-analysis and resort to MCTS only for addressing exceptional situations. One may point out that case-analysis is still dependent on training data, which is a potential conflict in our solution. Nonetheless, our experimental results demonstrated that it was not the case provided that the analysis structure is sufficiently *generic*. Algorithm 1 presents a simplistic overview of our AI design where $\mathcal{S}$ represents the given game state and $\mathcal{K}$ the

set of states covered by our case-analysis. The other 2 functions—SmartSearch (our case-analysis) and MCTS—will be described in subsequent subsections.

Algorithm 1 Hybrid AI Design

```
function GetBestAction(S)
    if S ∈ K then
        return SmartSearch(S)
    else
        return MCTS(S)
    end if
end function
```

4.1 Smart search

4.1.1 Generic

By smart search, we refer to a set of heuristics derived from our manual case-analysis. Those heuristics are used to handle some sets of states that have been classified by humans and help avoid using the expensive MCTS. Their only concern is to be *generic*. By that, it means they must be dynamic against changes in the environment's setting. What does it mean?

Algorithm 2 showcases an example of a non-generic heuristic where x_1 and x_2 represents the positions of the two players. This example suggests an attacking action when our distance to the opponent is below a certain threshold (123) or a movement action to get closer, otherwise. What if all actions have the attack range much less than 123. For example, if it is only 23, our player would perform meaningless attacks. This heuristic is non-generic and hence inapplicable in our problem where the environment setting is dynamic. How can we make it generic?

Algorithm 2 A Non-generic Heuristic

```
if |x1 − x2| < 123 then
    Attack
else
    Get closer
end if
```

Algorithm 3 presents an alternative way to rewrite the non-generic heuristic illustrated in Algorithm 2. In this example, $\mathcal{A}$ represents the set of all actions. By rewriting it this way, this heuristic can adapt to any changing environment since the limit M is computed at run-time. The computational power required for such a computation is trivial compared to that of MCTS. All dynamic heuristics like this can be generalized to all environments and can be considered generic.

Algorithm 3 A Generic Heuristic

```
M ← max_{a∈A} (range of a)
if |x1 − x2| < M/2 then
    Attack
else
    Get closer
end if
```

The rest of this subsection will describe two such generic heuristics: potential selection and defense-attack switch. The former searches for an excellent attack action and the latter determines if we should continue attacking the opponent or start defending ourselves. The combination of these 2 heuristics will result in our smart search strategy.

4.1.2 Potential selection

What is the best action? Is it the action that causes the opponent to suffer the most for now or one that can contribute to our future victory? This heuristic prioritizes the latter over immediate benefits from the former. The general idea is as follows:

1. Send the opponent to a *blocked* state.
2. Repeat step 1 before the opponent recovers.

A blocked state implies a state where it takes significant time for a player to recover. Before recovering completely, he cannot issue another action. By attacking nonstop in that fashion, we can completely take on the offensive while the other player can only passively receive our attacks. In other words, the action of the highest potential is one that is most likely to send the other player to a slow-recovering situation.

To implement that idea in the fighting game AI challenge, we define a *potential* for every action in accordance with its startup time (illustrated by Fig. 2) and its likelihood to send the opponent to a blocked state. The latter is self-explanatory. As for the former, the reason is to increase the likelihood that we can land an attack on the opponent before his full recovery. Algorithm 4 outlines this heuristic in greater detail.

Algorithm 4 Potential Selection Heuristic

```
function PotentialSelection(S)
    A* ← {a | a ∈ A and a can send the opponent to a blocked state}
    for a ∈ A* do
        if a cannot reach the opponent given S then
            Remove a from A*
        end if
    end for
    return argmin_{a∈A*} (startup time of a)
end function
```

4.1.3 Defense-attack switch

In a fight, attacking alone is not always a smart strategy. Another indispensable aspect lies in the timing decision of

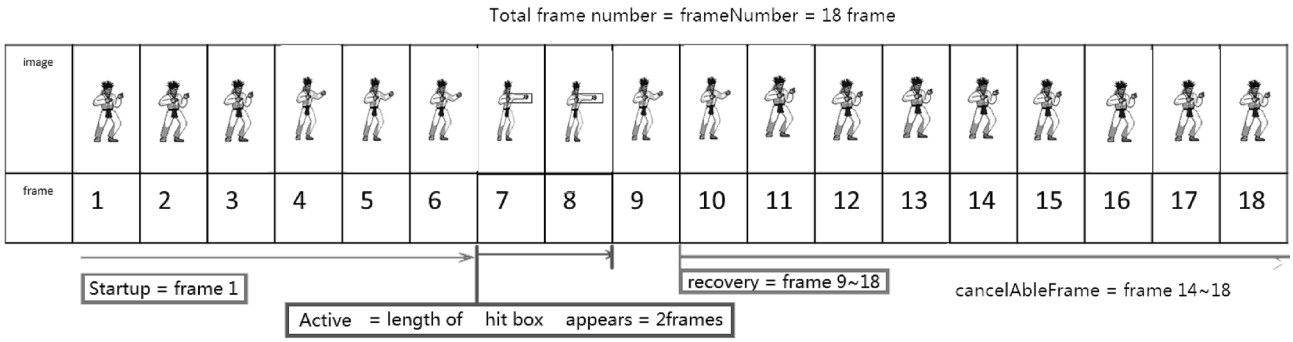

Fig. 2 Illustration of an action's phrases

whether to run away or attack. In this heuristic, we define an upper *HP threshold*, beyond which it is unlikely for the opponent to turn the tables. When our HP difference compared to the opponent exceeds that threshold, we will get into defense mode by choosing a runaway or far-range attack action. Otherwise is attack mode, in which our player will choose a suitable action to close the distance with the opponent as soon as possible and get ready for an attack. Since the threshold to determine whether we should attack or defend depends solely on the difference between our and the opponent's current HP levels and the exact value is unimportant as long as it is relatively high, this analysis is simple, non-innovative but unexpectedly effective as demonstrated by our competition results. Algorithm 5 outlines our implementation of this approach for the fighting game AI challenge in greater detail.

Algorithm 5 Defense-Attack Switch Heuristic

```
function DefenseAttackSwitch(x, y)
    T ← 1/2 × inital HP
    if x - y ≥ T then
        Apply defend mode
    else
        Apply attack mode
    end if
end function
```

4.2 Monte-Carlo tree search

4.2.1 Basic structure

Monte-Carlo tree search (MCTS) Coulom (2006) can easily be considered one of the most important breakthroughs in AI. Compared with traditional methods, MCTS is capable of gathering information from a simulated future via the means of random simulation. That whole learning process happens during run-time. A more thorough tutorial on MCTS for gaming AI can be found in Chaslot et al. (2008), Yoshida et al. (2016), Ishihara et al. (2016), Ishii et al. (2018), Ishihara et al. (2018), Vodopivec et al. (2017), Zooket al. (2019), and Demediuk et al. (2017). In this subsection, we would present the basic structure of MCTS via Algorithm 6, and propose an optimization which was implemented in our submission to the fighting game AI challenge. Their general intuition, nonetheless, can be generalized for wider applications.

In Algorithm 6, the budget represents the real-time constraints (16.67 ms in this challenge), $\mathcal{S}$ is the current game state, $\mathcal{A}$ is the collection of all actions, and ϕ the transition function for finding the next game state after performing an action. MCTS considers each game state as a node in a search tree. As outlined in the pseudocode, the search consists of 4 stages: selection, expansion, simulation, and back-propagation. The search target is the best action that can transform the input state $\mathcal{S}$ to some direct descendant with the highest long-term benefits.

Algorithm 6 Monte-Carlo tree Search

```
function MCTS(S)
    while budget remains do
        S' ← S
        while S' is fully expanded do                    ▷ Selection by UCB
            S* ← {φ(S', a) | a ∈ A}
            S' ← argmax_{s∈S*}(UCB(s))
        end while
        if S' is not end state then                      ▷ Expansion into a new state
            a ← an untried action
            S' ← φ(S', a)
        end if
        Simulate a random game from S'                   ▷ Simulation
        Update number of visits for all ancestors from S'  ▷ Back-propagation
    end while
    return a ∈ A where φ(S, a)) is most visited
end function
```

This algorithm utilizes random game simulations as a means to estimate the long-term value of a node. Nonetheless, it is unrealistic to perform simulations for all descendants since there are infinitely many of them. That is why a selection strategy using upper-confidence bound (UCB) algorithm which can search for the most worthy node is employed. A complete explanation for the mechanism behind UCB can be found in Lattimore and Szepesvári (2018) and Francisco-Valencia et al. (2019). In brief, the basic idea is that it works by balancing between exploitation

(a node with high success likelihood) and exploration (a node which is rarely visited). The mathematical formulation of that idea is presented in (1),

$$\bar{W} + c \times \sqrt{\frac{\log_2 N_p}{N_c}} \tag{1}$$

where $\bar{W}$ stands for the success likelihood such as the marginal increase of HP or energy compared to the opponent (exploitation term), N_p and N_c the number of visits in the parent and current nodes, respectively ($\sqrt{\frac{\log_2 N_p}{N_c}}$ is the exploration factor), and c the constant balancing the 2 terms.

Starting from a descendant with maximal UCB value, the algorithm expands into a random unvisited child and performs random game simulations from there. After that is back-propagation to update the current nodes and all its ancestors (their $\bar{W}$ and $\bar{N}$ values). The cycle repeats until our time limit is exceeded and the best action from the input state is one the leads to the most visited descendant.

As can be seen, since this algorithm is mainly based on information from simulations. It is domain-independent and can be applied for unlimited areas, not only fighting games. Nonetheless, the most important factor making it useful for a real-time AI challenge is its tolerance for resources. A higher budget (more time) can lead to more simulations and better results, but practice has demonstrated that MCTS works sufficiently well even for a moderate budget. Nonetheless, when the time allowance is way too low, even MCTS has to struggle, which is why we will propose a possible optimization for MCTS to make it work better for real-time fighting games.

4.2.2 Optimization

The key to our proposed optimization is to lower the consumption of memory, which is highly effective against AIs in a real-time environment. Due to real-time constraints, the number of simulations is relatively small. In this scenario, the object creation overhead will account for a significant amount of total running time. In other words, lowering memory creation, especially in this case, will significantly improve the overall execution time of MCTS, which is similar to the reason why insertion sort is faster for small input (Bingmann et al. 2020).

A possible improvement is to downsize the set of possible actions $\mathcal{A}$ as in Algorithm 6. Instead of considering all possible actions, it is possible to prioritize some over the others in a similar fashion shown in Algorithm 4. In other words, Algorithm 4 can be adapted to output n prioritized actions as shown in Algorithm 7 where ϵ is a constant that allows some randomness in the result.

Algorithm 7 Pseudocode for filtering prioritized actions

```
function GetPrioritizedActions(S, n)
    A* ← {a | a ∈ A and a can send the opponent to a blocked state}
    for a ∈ A* do
        if a cannot reach the opponent given S then
            Remove a from A*
        end if
    end for
    while |A*| < n + ε do
        Add a random action to A*
    end while
    while |A*| > n do
        Remove a random action from A*
    end while
    return A*
end function
```

Another implementation-specific optimization is to avoid extra memory consumption from keeping track of tried and untried actions for each game state (as in the expansion part of Algorithm 6). Usually, most developers would assign 2 action lists for each node, which is a waste of memory. The best course of action is to combine them into 1 single list and decrease the memory consumption by 50%. A realization of this idea is presented in Algorithm 8 where L is the list of actions and $0 <= m < |L|$ the barrier, before which are tried actions and after which are untried actions.

Algorithm 8 Pseudocode for keeping track of tried and untried actions

```
L ← {untried actions}
m ← 0
function TryAction(i)
    ...
    Swap L[i] and L[m]
    m ← m + 1
end function
```

5 Experiments and evaluation

As mentioned previously, our research was conducted via the fighting game AI challenge. Therefore, the competition results serve as our experimental results. Table 1 showcases the distinction of our solution before and after adapting our hybrid design. The left half of the table represents our mid-term result against 5 other competitors in which our optimizations were still absent. Albeit we did not have high expectations, it was shocking to be at the bottom of the ranking and to make it worse, we were losing every single match. That painful result, nonetheless, served well as a step to enhance our improvement as shown in the second half of the table. From the bottom, we climbed straight to

Table 1 Performance matrix

Category	Midterm (before)		Final (after)	
Standard	Win matches	Ranking	Win matches	Ranking
	0	6/6	39	1/9
Speed running	Win time	Ranking	Win time	Ranking
	70 s	6/6	52.72 s	1/9

Table 2 Performance stability

Solution	Standard	Speed running	Ranking change
Our solution	1	1	No change
Thunder	2	5	−3
SampleMctsAi	3	3	No change
MogakuMono	4	2	+2
JayBot_GM	5	4	+1
SimpleAI	6	7	−1
UtalFighter	7	6	+1
MultiHeadAI	8	8	No change
BCB	9	8	+1

the top against a higher number of opponents (with new participants) and in a record amount of time. These results demonstrated our performance.

Another imperative factor to assess our solution is stability which is demonstrable by in Table 2. As can be seen, most participants experienced certain instability for their solution. Most solutions are either too unstable or too terrible, except for SampleMctsAi whose performance is still far beneath ours. That demonstrates the fact that our design is the only stable solution that well-performed (first place for all).

6 Conclusion and future work

This paper contributes a scalable algorithmic design for real-time AI which has proven very successful for real-time fighting games in a special environment like LUD where most training-dependent methods are unusable. Our results demonstrated that a proper blend between simple case-analysis and Monte Carlo tree search could produce an excellent AI under real-time constraints. Besides, we also propose various generic heuristics and general optimization in terms of memory consumption for Monte-Carlo tree search. Those are generic ideas that should be applicable in other AI gaming problems such as Ms. Pacman Challenge (Williams et al. 2016) or the General Video Game AI (Perez-Liebana et al. 2016), not only restricted to the Fighting Game AI Challenge.

As a next step, we will continue to implement the heuristics presented in this paper for a wider range of domains and prove the generalization of our ideas. Also, our team is aware of the recent trends in data-based learning methods such as deep reinforcement learning. They have always been the primary motivation for AI development but were, unfortunately, not suitable for this type of challenge. The search for effective use of those methods in LUD-like environments where critical data is not available for training in advance will serve well as the target for our future research.

Last, but not least, another possibility is to improve the performance of MCTS. What if the simulations required by MCTS are not random but are based on certain heuristics? Will improving the quality of simulations decrease the number of simulations needed? Albeit this is a non-trivial approach, its applications are far beyond the scope of fighting game AI and will be the perfect direction for this research to evolve.

Acknowledgements We appreciate the support of the Frankfurt University of Applied Sciences during the implementation of this project and the Intelligent Computer Entertainment Lab from Ritsumeikan University for their awesome fighting game AI platform which was indispensable for conducting our experiments.

References

Back T (1996) Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms. Oxford University Press, Oxford

Balabanov K, Logofătu D (2019) Developing a general video game ai controller based on an evolutionary approach. In: Asian conference on intelligent information and database systems. Springer, pp 315–326

Bingmann T, Marianczuk J, Sanders P (2020) Engineering faster sorters for small sets of items. arXiv:2002.05599

Campbell M, Hoane AJ Jr, Hsu F (2002) Deep blue. Artif Intell 134(1–2):57–83

Cavazza M (2000) Al in computer games: survey and perspectives. Virtual Real 5(4):223–235

Chaslot G, Bakkes S, Szita I, Spronck P (2008) Monte-Carlo tree search: a new framework for game ai. In: AIIDE

Cho BH, Jung SH, Seong YR, Oh HR (2006) Exploiting intelligence in fighting action games using neural networks. IEICE Trans Inf Syst 89(3):1249–1256

Coulom R (2006) Efficient selectivity and backup operators in Monte-Carlo tree search. In: International conference on computers and games. Springer, pp 72–83

Demediuk S, Tamassia M, Raffe WL, Zambetta F, Li, X, Mueller F (2017) Monte Carlo tree search based algorithms for dynamic difficulty adjustment. In: 2017 IEEE conference on computational intelligence and games (CIG). IEEE, pp 53–59

Francisco-Valencia I, Marcial-Romero JR, Valdovinos-Rosas RM (2019) Some variations of upper confidence bound for general game playing. In: Mexican conference on pattern recognition. Springer, pp 68–79

Harper T (2013) The culture of digital fighting games: performance and practice. Routledge, Abingdon

Ishihara M, Miyazaki T, Chu CY, Harada T, Thawonmas R (2016) Applying and improving Monte-Carlo tree search in a fighting game ai. In: Proceedings of the 13th international conference on advances in computer entertainment technology, pp 1–6

Ishihara M, Ito S, Ishii R, Harada T, Thawonmas R (2018) Monte-Carlo tree search for implementation of dynamic difficulty adjustment fighting game ais having believable behaviors. In: 2018 IEEE conference on computational intelligence and games (CIG). IEEE, pp 1–8

Ishii R, Ito S, Ishihara M, Harada T, Thawonmas R (2018) Monte-Carlo tree search implementation of fighting game ais having personas.

In: 2018 IEEE conference on computational intelligence and games (CIG). IEEE, pp 1–8
Kim MJ, Kim KJ (2017) Opponent modeling based on action table for mcts-based fighting game ai. In: 2017 IEEE conference on computational intelligence and games (CIG). IEEE, pp 178–180
Kim MJ, Ahn CW (2018) Hybrid fighting game ai using a genetic algorithm and Monte Carlo tree search. In: Proceedings of the genetic and evolutionary computation conference companion, pp 129–130
Kim MJ, Kim JS, Lee D, Kim SJ, Kim MJ, Ahn CW (2019) Integrating agent actions with genetic action sequence method. In: Proceedings of the genetic and evolutionary computation conference companion, pp 59–60
Lattimore T, Szepesvári C (2018) Bandit algorithms, p 28
Logofatu D, Leon F, Muharemi F (2019) General video game ai controller-integrating three algorithms to bring a new solution. In: 2019 23rd international conference on system theory, control and computing (ICSTCC). IEEE, pp 856–859
Lu F, Yamamoto K, Nomura LH, Mizuno S, Lee Y, Thawonmas R (2013) Fighting game artificial intelligence competition platform. In: 2013 IEEE 2nd global conference on consumer electronics (GCCE). IEEE, pp 320–323
Martinez-Arellano G, Cant R, Woods D (2016) Creating ai characters for fighting games using genetic programming. IEEE Trans Comput Intell AI Games 9(4):423–434
Nakagawa Y, Yamamoto K, Thawonmas R (2014) Online adjustment of the ai's strength in a fighting game using the k-nearest neighbor algorithm and a game simulator. In: 2014 IEEE 3rd global conference on consumer electronics (GCCE). IEEE, pp 494–495
Nakagawa Y, Yamamoto K, Yin CC, Harada T, Thawonmas R (2015) Predicting the opponent's action using the k-nearest neighbor algorithm and a substring tree structure. In: 2015 IEEE 4th global conference on consumer electronics (GCCE). IEEE, pp 533–534
Ng BW (2006) Street fighter and the king of fighters in Hong Kong: a study of cultural consumption and localization of Japanese games in an Asian context. Game Stud 6(1):2006
Nork B, Lengert GD, Litschel RU, Ahmad N, Lam GT, Logofătu D (2018) Machine learning with the pong game: a case study. In: International conference on engineering applications of neural networks. Springer, pp 106–117
Osaka S, Thawonmas R, Shibazaki T (2014) Investigation of various online adaptation methods of computer-game ai rulebase in dynamic scripting. In: Proceedings of the 1st international conference on digital interactive media entertainment and arts (DIMEARTS 2006) (2006)
Perez-Liebana D, Samothrakis S, Togelius J, Schaul T, Lucas SM (2016) General video game ai: competition, challenges and opportunities. In: 13th AAAI conference on artificial intelligence
Pinto IP, Coutinho LR (2018) Hierarchical reinforcement learning with monte carlo tree search in computer fighting game. IEEE Trans Games 11(3):290–295
Robison AD (2017) Neural network ai for fighting ice. California Polytechnic State University, San Luis Obispo (Thesis)
Silver D, Huang A, Maddison CJ, Guez A, Sifre L, Van Den Driessche G, Schrittwieser J, Antonoglou I, Panneershelvam V, Lanctot M et al (2016) Mastering the game of go with deep neural networks and tree search. Nature 529(7587):484
Silver D, Schrittwieser J, Simonyan K, Antonoglou I, Huang A, Guez A, Hubert T, Baker L, Lai M, Bolton A et al (2017) Mastering the game of go without human knowledge. Nature 550(7676):354–359
Skinner G, Walmsley T (2019) Artificial intelligence and deep learning in video games a brief review. In: 2019 IEEE 4th international conference on computer and communication systems (ICCCS). IEEE, pp 404–408
Tamassia M (2017) Artificial intelligence techniques towards adaptive digital games. College of Science, Engineering and Health, RMIT University
Taylor ME, Carboni N, Fachantidis A, Vlahavas I, Torrey L (2014) Reinforcement learning agents providing advice in complex video games. Connect Sci 26(1):45–63
Thawonmas R, Osaka S (2006) A method for online adaptation of computer-game ai rulebase. In: Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology, p 16
Thuan LG, Logofătu D, Badică C (2019) A hybrid approach for the fighting game ai challenge: balancing case analysis and Monte Carlo tree search for the ultimate performance in unknown environment. In: International conference on engineering applications of neural networks. Springer, pp 139–150
Vodopivec T, Samothrakis S, Ster B (2017) On monte carlo tree search and reinforcement learning. J Artif Intell Res 60:881–936
Williams PR, Perez-Liebana D, Lucas SM (2016) Ms. pac-man versus ghost team cig 2016 competition. In: 2016 IEEE conference on computational intelligence and games (CIG). IEEE, pp 1–8
Yoon S, Kim KJ (2017) Deep q networks for visual fighting game ai. In: 2017 IEEE conference on computational intelligence and games (CIG). IEEE, pp 306–308
Yoshida S, Ishihara M, Miyazaki T, Nakagawa Y., Harada T, Thawonmas R (2016) Application of Monte-Carlo tree search in a fighting game ai. In: 2016 IEEE 5th global conference on consumer electronics. IEEE, pp 1–2
Zook A, Harrison B, Riedl MO (2019) Monte-Carlo tree search for simulation-based strategy analysis. arXiv:1908.01423
Zuin GL, Macedo YP, Chaimowicz L, Pappa GL (2016) Discovering combos in fighting games with evolutionary algorithms. In: Proceedings of the genetic and evolutionary computation conference, pp 277–284