A Hybrid Approach for the Fighting Game AI Challenge: Balancing Case Analysis and Monte Carlo Tree Search for the Ultimate Performance in Unknown Environment

Lam Gia Thuan[1], Doina Logofatu[2] and Costin Badica[3]

[1] Vietnamese-Germany University, Faculty of Engineering, Le Lai, Thu Dau Mot, Vietnam
[2] Frankfurt University of Applied Sciences, Department of Computer Science and Engineering, Nibelungenpl. 1, 60318 Frankfurt, Germany
[3] University of Craiova, Department of Computer Sciences and Information Technology, 200285 Craiova, Romania
logofatu@fb2.fra-uas.de

Abstract. The challenging nature of the Fighting Game AI Challenge originates from the short instant of response time which is a typical requirement in real-time fighting games. Handling such real-time constraint requires either tremendous computing power or a clever algorithm design. The former is uncontrollable by the participants, as for the latter, the competition has received a variety of submissions, ranging from the naivest case analysis approach to those using highly advanced computing techniques such as Genetic Algorithms (GA), Reinforcement Learning (RL) or Monte Carlo Tree Search (MCTS), but none could provide a stable solution, especially in the LUD division, where the environment setting is unknown in advance. Our study presents our submission to this challenge in which we designed a winning solution in the LUD division which, for the first time, stably outperformed all players in all competition categories. Our results demonstrate that a proper blend of simple case analysis and advanced algorithms could result in the ultimate performance.

Keywords: Fighting Game AI Challenge, real-time fighting game, LUD division, case analysis, MCTS.

1 Introduction

Advances in computing have given rise to computer game complexity, to the extent that human ability is no longer sufficient to handle the vast number of game states. Human gamers will soon be dominated by computer programs due to the emergence of powerful techniques such as Monte Carlo Tree Search [1] and Machine Learning [2] in which the programs can learn the solution with little human intervention. Aided by modern computer power, these algorithms have demonstrated a capacity beyond that of human experts with numerous examples such as Deep Blue [3] or AlphaGo [4], but does that imply that human guidance is completely irrelevant in this day and age?

The success of programs like AlphaGo was partially thanks to the nature of the game, which was chess, a turn-based game, in which modern computers are given sufficient time to process. However, in a real-time environment where programs must response reasonably well within a short instance of time, current processing power proves insufficient since modern game states can grow exponentially while processors are increasingly closer to their physical limit [5].

This paper presents our submission to the Fighting Game AI challenge [6] 2018, a challenging AI competition that aims at solving the general real-time fighting problem. Our research focuses on the design of an algorithm that can handle the vast number of game states in a reasonable way given our average computational power. Our algorithm is a proper blend between a generic case analysis approach with the winning MCTS algorithm. Our results show that by blending some human wisdom with MCTS, the resulting performance is superior to all in existence.

2 The Fighting Game AI Challenge

2.1 Overview

The Fighting Game AI Challenge is initiated by the Intelligent Computer Entertainment Lab of the Ritsumeikan University to promote AI research towards general fighting games, a classical class of games, in which players compete against each other until only one remains using techniques resembling those in martial arts. Starting since 2013, the competition has been well-received by scholars around the world and achieved a high reputation among the most prestigious academic conferences worldwide, including the IEEE Conference on Computational Intelligence and Games (CIG). CIG 2018 marked another successful milestone of the challenge with the emergence of numerous interesting solutions, among which was our submission, the first that could perform well and stably in the most challenging LUD division in all competition categories.

2.2 Organization and Rules

The competition consists of 3 divisions – ZEN, GARNET and LUD, each is further subdivided into 2 categories: STANDARD (participants fighting each other) and SPEED-RUNNING (participants defeating the organizer's program in the shortest amount of time). Each player is given a set of actions, from which one must be selected within a fixed unit of time, namely frame. The player with the most HP remained after at most 3600 frames is the winner of the game. In the STANDARD category, the player that wins the most games, is the final winner, while in SPEED-RUNNING, the final winner is the player that wins against a common opponent in the least number of frames.

2.3 Our focus - The LUD division

The LUD division distinguishes itself from ZEN and GARNET by hiding all action data in advance. An action is characterized by a number of parameters, a subset of which can be shown in Table 1. These parameters are vital for memorized methods such as Q-Learning or if-else approach, in which participants can train their program for a long time before the contest and store learned experiences into files that will be retrieved during the playtime, which is a realistic approach for the first 2 divisions in that they have all information published prior to the competition. Whilst for the LUD division, such essential information is only provided at the start of the game as a step to make it closer to the general fighting problem, which, at the same time, prevents memorized approaches from emerging victorious. That is where self-learning methods such as MCTS reign considering that they have no need for prior training.

Table 1. A Sample Subset of Action Parameters.

Parameter Name	Description
Frame Number	The number of frames required to perform.
Horizontal Speed	The horizontal velocity.
Vertical Speed	The vertical velocity.
Hit Area	The impact area as a rectangle.
State	The state of opponent after getting hit.

MCTS is, unfortunately, not without its limitations. No prior training implies the need for self-learning during the playtime which is not favorable in a real-time environment. Albeit the majority of the submissions were dominated by MCTS, only those who could manage to reduce its time-consuming nature could become victors. The most successful example being Eita Aoki [7]. By discovering a winning heuristic, he reduces the use of MCTS to the minimum possible and became the 3-time consecutive winner of this challenge. Nonetheless, even his solution is unable to conquer the LUD division since his heuristic could not be formularized without prior information.

Our submission proposed the first stable winning solution to the challenging LUD division. Albeit missing a detailed case analysis made us stay only in the second place in the first 2 divisions, our contribution is still of paramount importance since LUD is, among the three, the closest division to the general real-time fighting problem.

3 Previous Work

The first period between 2013 and 2015 could be regarded as the Dark Ages in the history of the competition in that submissions were entirely populated with variants of the if-else approach with little to no sign of new directions. Regardless, there were numerous interesting heuristics, as shown in Table 2, which are still relevant until now.

Table 2. Competition Notable Approaches in 2013-2015 period.

Year	Approach
2013	Position Analysis + Random Limited Choices

	Position Analysis + Fixed Choices + Defending/Escaping Heuristic
2014	Simple Reinforcement Learning for memorizing states Case Analysis + Opponent Modelling Simple Fuzzy Logic
2015	Position Analysis + Random Limited Choices + Runaway Case Simple Machine Learning for memorizing states

The initial popularity of these if-else variants is understandable since advanced techniques require a certain level of design and implementation skill and algorithm design is too hard of a field even for experienced experts, making them unpopular among participants, most of whom are students or young professionals. Case analysis with heuristics is much easier to implement and hence was the only technique in use in the first year of the competition. Subsequent years saw more advanced techniques such as Reinforcement Learning or Fuzzy Logic, but their implementation was still too simplistic to produce satisfiable results. Worse came to worst, they were even losing against the naïve and much-simpler-to-implement if-else approaches.

In 2016, the organizers introduced MCTS into the competition as a sample, which marked a new standard for submissions and resulted in the emergence of more sophisticated solutions. Some of the most successful participants can be mentioned as follows:

- Eita Aoki for combining his winning unbreakable corner heuristics with MCTS which overwhelmingly dominated the first 2 divisions: LUD and GARNET.
- Youssouf Ismail Cherifi for combining his consecutive strike heuristics with MCTS.
- Man-Je Kim and Kyung-Joong Kim for the first evolutionary algorithm solution in combination with MCTS [8].

In addition, many submissions reimplemented algorithms that had been seen in previous years, but with much more mature implementation. Albeit they are unable to compete with MCTS and evolutionary algorithms, they have demonstrated their true worth by at least outperforming simplistic case analysis approaches.

Despite all these advances, the LUD division remained unmanageable until our participation. In 2018, we proposed a solution that outperformed all others in LUD divisions in all categories and became the first stable winning solution to this division.

4 The winning solution to the LUD division

Careful analysis of the previous solutions leads us the conclusion that a successful solution is the one that is based on MCTS but does not abuse it, as illustrated in the following pseudocode:

```
Action findBestAction (GameState state) {
  Action action = intelligentSearch(state)
  if (action != null)
    action = MCTS(state)
  return action
}
```

In detail, a successful solution should implement an intelligent search method in addition to MCTS and prioritize it whenever possible.

For the first 2 divisions, ZEN and GARNET, the distinction between submitted MCTS-based solutions usually lies in the former and almost all participants employed the standard implementation of MCTS provided by the organizer. Unfortunately, none could provide a similar solution for the LUD division, since the lack of data prevented participants from formularizing a working heuristic. Our study concludes that such a model is still applicable to the LUD division as long as the heuristic is made *generic*.

4.1 Adaptive Intelligent Search

This is the winning aspect of our solution but also the simplest one in that our analysis is truly generic - almost independent of actual values, which is why it works well in this unknown context. The search for the best action in our solution depends on the following factors: potential, preselection, and upper distance analysis.

Potential. To analyze each action's potential, it is of paramount importance to analyze parameters outside those mentioned in Table 1, including the startup time and the resulting state of the player under attack. The latter is more prioritized than the former in our submission since startup time does not vary much among different actions.

Startup time. The complete process for performing an attack consists of multiple periods, including a startup, active, recovery and canceling period, as visualized in Fig. 1.

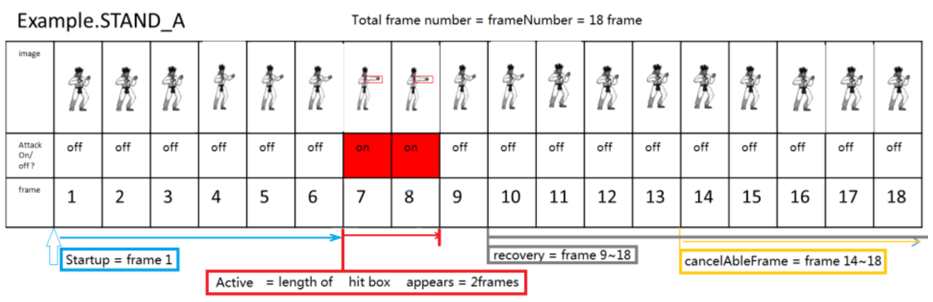

Example.STAND_A
Total frame number = frameNumber = 18 frame
image
Attack On/ off ?
off off off off off off on on off off off off off off off off off off
frame
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
Startup = frame 1
Active = length of hit box appears = 2frames
recovery = frame 9~18
cancelAbleFrame = frame 14~18

Fig. 1. Motion Illustration (The Fighting Game AI Challenge)

Among which, we believe that startup is the most imperative period. The rationale behind this decision is due to the concurrent nature of the problem as illustrated in Fig. 2.

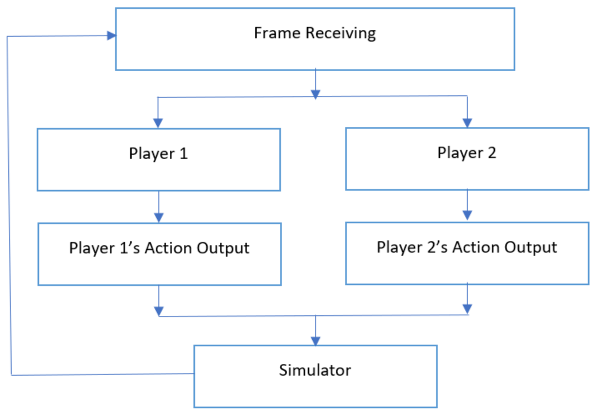


Fig. 2. Game Processing Flow.

As illustrated in Fig. 2, players perform their respective action at the same time, implying the possibility that one player can be put under attack even before he or she is ready to act. Thus, the faster the attack can start, the less likely our player will end up in a disadvantageous situation, which was why this factor should be considered as a priority to determine the potential of the action.

Resulting state of the player under attack. Nonetheless, fast startup alone does not guarantee an excellent action since the opponent still has a chance to revenge in the next attack. A promising attack should be one that opens more opportunity for future attacks. In further details, it means that a good attack should lead the opponent into a vulnerable state in which it is difficult or even impossible for him or her to block the next one, forming a series of consecutive moves, namely combo [9]. In this particular challenge, the most known vulnerable state is the DOWN state – a state where the opponent is completely knocked out and became unable to retaliate. Hence actions that can result in the opponent's DOWN state will be our first-class priority.

Preselection. A further improvement is the preselection of a minimum set of actions based on their potential for speed improvement during the playtime. After that, an additional reselection to further minimize the selected list will be conducted based on random game simulation and decide which actions to choose according to its respective total damage for the opponent during the random games. An illustration can be shown as follows.

```
List<Action> preselection(Int round1Size, Int round2Size)
{
  List<Action> firstRound = getAllActions()
    .sortedAscBy (action -> action.getPotential())
    .toList()
    .subList(0, round1Size)
  Map<Action,Int> damageByAction =
                 simulateRandomGamesAndRecord(firstRound)

  List<Action> secondRound = firstRound
    .sortedDescBy (action -> damageByAction[action])
    .toList()

  return secondRound
}
```

Upper Distance Analysis. Attacking is important, but another indispensable aspect of victory in a fighting game is the control of player movement, which usually replies on actual values. In order to make it generic to work in the LUD division, we define a much simpler strategy: attack or defend. In attack mode, our player will choose actions that get us as close to the opponent as possible. In defend mode, we will just choose actions that run away from the opponent. The threshold to determine whether we should attack or defend depends solely on the difference between our current HP and the opponent HP as illustrated in the pseudocode below. This will be executed before all to filter irrelevant actions before selection.

```
List<Action> upperDistanceAnalysis() {
  if (myHP - opponentHP >= UPPER_LIMIT) {
    return defendActions() // actions that runs away
  } else {
    return attackActions() // actions that get us closer
  }
}
```

The threshold is set to be an upper bound value, which can be *any high enough* value that if it is met, we are certainly going to win regardless of what happens afterward. Since such value is independent of the action data, it can be chosen generically.

In addition to the above factors, each action will only be considered for selection if the current amount of energy permits it. Actions that require too much energy will be filtered beforehand. Last, but not least, a simple minimax algorithm will be used to select those with the same priority and each action will only be selected if it can produce a positive impact on the opponent. The priority to select an action will be first, in those chosen in the preselection, followed by those with the largest potential.

4.2 A brief overview of our optimized MCTS

As stated in subsection 4.1, the core of our contribution is the intelligent search, not the MCTS since almost all MCTS-based solutions were just reusing the provided sample MCTS code with little to no improvement. Moreover, the sample provided implementation is a very standard one, hence an extensive description will not be included in our paper. Further information can be found at [10]. Instead, in this section, we will just briefly review the key points of MCTS used in our program and briefly introduce our optimization.

Monte Carlo Tree Search. MCTS is a simulation-based search algorithm. Like other algorithms, first it will search for the known *best* node, then it will extend the search tree from there. The best node is chosen in a way that balances both the depth (exploitation) of the search tree and breadth (exploration) of opportunities and the way to achieve that is to use the famous Upper Confidence Bound (UCB) formula. The full form of UCB used for computing the potential of each node in our solution is as follows.

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

, in which $\overline{W}$ is the average number of wins, N_p is the number of visits in the direct parent node and N_c is the number of visits in the current node. $c = \sqrt{2}$ is the exploration parameter that is used to balances between exploitation ($\overline{W}$) and exploration ($\sqrt{\frac{\log_2 N_p}{N_c}}$).

In the context of our solution, each node is a game state. The root node is the current game state and all others are virtual nodes that will be created by game simulation. Each new node is created from the current node when we try to perform a different action. The tree branch will go deep until we can decide if the current state is a win or loss and then we can recursively recompute all information such as the number of wins and the number of visits from the current state up to root. The entire process is repeated until time runs out (1 frame = $\frac{1}{60}$ second). The selected action is one that leads to the most visited direct child from root. The complete procedure can be summarized as follows.

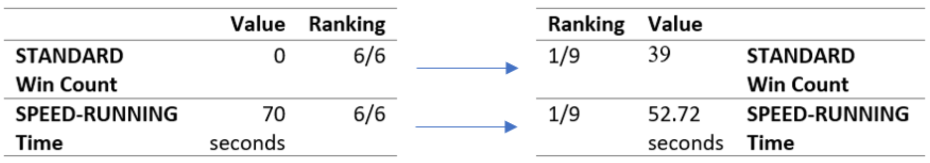
```
Action MCTS(Node rootNode) {
  Node node = rootNode
  while (node.isNotEndState()) (
    if (node.areAllActionConsidered()) {
      node = findBestChildNodeByUCB(node)
    } else {
      Action action = node.pickAnyActionNotConsidered()
      node = createNode(node, action)
    }
  }
  while (node != null) {
    updateNumberOfWins(node)
    updateNumberOfVisists(node)
    node = node.getParent()
  }

  return findMostVisistedChild(rootNode).getAction()
}
```

Optimization. Our improvement lies in the reduction in memory, which eventually leads to speed improvement as in the case similar to why insertion sort is faster than quick-sort for small lists of elements [11]. Each node in a standard implementation needs to keep track of both the actions that we have considered (simulated) and actions we have not, implying that we may require two arrays for storing them. That approach is used by most people since it is both intuitive and simple to implement. In our submission, we avoid the creation of the additional array by keeping track of only the number of unused actions. Every newly used action will be swapped in $O(1)$ with the first action that is not yet used. By utilizing this trick, we manage to reduce the memory consumption by half and since the number of possible actions after various filters in subsection 4.1 is relatively small, the performance is significantly enhanced. Unfortunately, this is not key to our success due to the choice of our programming language which will be explained in section 5.

5 Evaluation

5.1 Performance

The first priority to evaluate a solution is naturally its performance. Fig. 3 illustrates the improvement of our solution before and after mixing our case analysis approach.



	Value	Ranking
STANDARD Win Count	0	6/6
SPEED-RUNNING Time	70 seconds	6/6

Ranking	Value	
1/9	39	**STANDARD Win Count**
1/9	52.72 seconds	**SPEED-RUNNING Time**

Fig. 3. Performance before and after applying our case analysis approach

The table on the left of Fig. 3 is our mid-term result against 5 other competitors in which we did not introduce the intelligent search. Albeit we did not have high expectation, it was still shocking that we were at the bottom of ranking in the LUD division at the beginning and even worse, we were knocked out by every other participant in every single game. However painful it was, it served well for us as a crucial step to highlight the brilliantness of our improvement as shown in the second table – our final result against 8 other participants, in which we climb straight to the top from the bottom. Being on the top of the ranking in both STANDARD and SPEED-RUNNING leagues demonstrate our performance.

5.2 Stability

The next key factor to evaluate our solution is stability which can be illustrated in Table 3, showing instability among other participants.

Table 3. Ranking Instability.

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

As can be seen from Table 3, most participants either perform too unstably or badly. Since SampleMctsAi belongs to the organizers, it is counted, implying that ours is the only stable solution that well-performed (first place for all).

5.3 Remarks about MCTS Optimization

As mentioned in the previous section, our solution is not only with one improvement – the intelligent search, but it also includes an optimization in MCTS. This subsection answers the question: Does it improve anything and why is it not key to the success of our solution?

The problem originates from our choice of programming language for implementing this solution - Kotlin [12], a promising programing language on JVM, while other participants all use Java, a much more mature language. In our submission, we chose Kotlin for its expressiveness and beauty, but it was unexpected that Kotlin compiler is still too young to generate bytecode of the same quality as Java compiler. In our experiment after the contest, we translated the Java programs into equivalent Kotlin code and were surprised to see that the Kotlin version lost every single match despite the algorithm in use is the same. Therefore, even though our MCTS implementation is of higher quality compared to others, it is invisible from the competition results.

Does our optimization really work? In our experiment, we compared MCTS programs with and without the optimization. With our optimization, it was still on the losing side but could handle approximately 1 out of every 3 matches instead of failing every single time as the program without optimization. Regardless, this result demonstrates that without our case analysis, we would have lost also in LUD division and that our intelligent search is truly effective.

6 Conclusion

In this study, we have contributed a promising solution for the general fighting problem via our submission to the Fighting Game AI Challenge 2018. Our solution is combination of case analysis and Monte Carlo Tree Search that produces stable and remarkable results in the LUD division, the division closest to the general real-time fighting game. Unfortunately, we were unable to make a significant optimization in the MCTS and still rely on human wisdom to generate the heuristics for the intelligent search which, at the same time, opens a new direction for our future research.

One possible direction for the future is to automate the generation of generic heuristics and the second is to further optimize MCTS for solving this challenge. One promising possibility for the former is the use of Deep Learning and Neural Networks to learn and formularize the heuristics before the competition, but the challenge of such approaches is to construct a model that can ensure the genericness of the resulting heuristics, meaning that it must be independent of actual values. Another approach to handle the latter is to introduce memorization into MCTS as illustrated in the AlphaGo Zero paper [13], which may significantly improve both the quality and performance of the search tree. The downside is that such an approach may be incompatible since the performance of the Google machines running AlphaGo Zero is still largely outperforming that of normal computers. Regardless, these are all promising directions, in which research towards real-time fighting games can evolve.

References

1. Chaslot, G., Bakkes, S., Szita, I., Spronck, P.: Monte-Carlo Tree Search: A New Framework for Game AI. AIIDE, The AAAI Press (2008)
2. Nork, B., Lengert, D., Litschel, R, Ahmad, N, Lam, G. T., Logofatu, D.: Machine Learning with the Pong Game: A Case Study. EANN 2018, pp. 106-117 (2018).
3. Campbell, M., Hoane, J., Hsu, F.: Deep Blue. Artificial Intelligence (2002).
4. Silver, D. et al: Mastering the game of Go with deep neural networks and tree search. Nature, 529 (7587), pp. 484-489 (2016).
5. ‘Are processors pushing up against the limits of physics?’, https://arstechnica.com/science/2014/08/are-processors-pushing-up-against-the-limits-of-physics, last accessed 21 Feb. 2019.
6. Fighting Game AI Competition, http://www.ice.ci.ritsumei.ac.jp/~ftgaic/index-1.html, last accessed 21 Feb. 2019.
7. 2018 Fighting Game AI Competition, https://www.slideshare.net/ftgaic/2018-fighting-game-ai-competition?ref=http://www.ice.ci.ritsumei.ac.jp/~ftgaic/index-R.html, last accessed 21 Feb. 2019.
8. Man-Je, K., Chang A: Hybrid fighting game AI using a genetic algorithm and Monte Carlo tree search. GECCO 18 Proc. of the Genetic and Evolutionary Computation Conference Companion, Kyoto, Japan (2018).
9. Zuin, G., Macedo, Y., Chaimowicz, L., Pappa, G.: Discovering Combos in Fighting Games with Evolutionary Algorithms. GECCO'16, pp. 277-284, Denver, CO, USA (2016).
10. James, S., Konidaris, G., Rosman: An Analysis of Monte Carlo Tree Search. Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17) (2017).
11. Heineman, G., Pollice, G., Selkow, S.: Algorithms in a Nutshell. O'Reilly Media. 2016.
12. Jangid, M: Kotlin – The unrivaled android programming language lineage. Imperial Journal of Interdisciplinary Research 3 (2017).
13. Silver, D. et al: Mastering the game of Go without human knowledge. Nature 550 (2017).