AI STRATEGY FORMULATION ACROSS VARIOUS GAME GENRES - A LITERATURE REVIEW

Cheuk Man Chan[1] and Robert Haralick[2]

[1]Computer Science Department, CUNY Graduate Center, New York, USA

[2]Distinguished Professor, Emeritus, Computer Science Department, CUNY Graduate Center, New York, USA

ABSTRACT

Game AI is a very active area of study in recent times. These advancements appear to coincide with advancements in processing power of both computers and gaming consoles, allowing for deploying more computationally intensive algorithm. This paper samples a small selection of papers from two periods, the late 2000s to early 2010s and the beginnings of the 2020s to current time in order to demonstrate this transition toward higher level of AI development.

KEYWORDS

Literature Review, Strategy Formulation, Game Playing Algorithm

1. INTRODUCTION

Game AI has been a part of video games since the beginning, controlling the behavior of the opposition to the player. However, it can be argued that the early stages of game AI does not reflect any degree of intelligence, but rather a mechanism that ensures both the player and the computer act within the rules of the game. At most such systems only determine the behavior of the computer using simple heuristic functions. As advancements in both hardware and software are made, the types of games along with their game AI systems gradually become more complex.

It is not until around the turn of the century where the beginning of true AI with machine learning techniques being introduced to both academic research and commercial development. While it is a given that commercial developments are focused on specific games, academic research is divided between specific games and general game playing. In fact, general game playing competitions are held regularly where participants submit a singular program that aims to play a large variety of games. This paper will review a small sample of papers focusing on both game-specific AI as well as general game playing AI. The papers are also drawn from two time periods to demonstrate the perpetuating advancement in the complexity of the algorithm.

2. GAME-TREE SEARCH WITH ADAPTATION IN STOCHASTIC IMPERFECT-INFORMATION GAMES

The algorithms presented in this section are a summary of a series of papers by Billings et al.: *Lokibot* [1], *Loki* [2], *Poki* [3, 4], *PsOpti* [5], and finally *Vexbot* [6]. The algorithms use poker, specifically limited Texas Hold'em poker, as the testing ground for developing an artificial intelligence program capable of competing with human players in an imperfect knowledge setting with multiple adversarial agents.

Lokibot, presented in Billings et al. [1], only focuses on the information known to the program, specifically, the cards in the program's hand and the community cards that is available to all the

players. Given that the program gives no considerations to the play style of the opponents, the program's own play style is rendered static and easily exploitable by an experienced player.

Loki, presented in Billings et al. [2], maintains a record of the opponent's decision making and the strength and potential of the opponent's hand when the decisions are made, then estimates the strength and potential of the opponent's current hand based on the decisions made by the player during the current game and adjusts its own decision making based on that estimation.

Poki, an algorithm presented by Billings et al. [3, 4], adds to the overall approach by introducing two additional features, previous actions and previous amounts to call. These features were identified by applying an artificial neural network over records of games played previously as the authors noted at the time that "the technique itself cannot be incorporated into a real-time system" due to the limits of the processing powers of machines at the time the research was conducted.

PsOpti presented in Billings et al. [5], makes use of the concept of Nash Equilibrium to determine the best strategy. The result, often called the minimax solution for its attempt to minimize the maximal loss, is a very defensive approach to the problem which, if applied by itself, offers the player no incentive to deviate from such a strategy. Beyond that point, Nash Equilibrium assumes the existence of the "rational" opposition player, which plays the same strategy and not mistakenly plays a move that returns a lesser payout. As the author points out himself, such an assumption "is definitely not the case in real poker, where the opponents are highly fallible." In addition, the program has an additional weakness in that it "is only an approximation of an equilibrium strategy, and it will not be feasible to compute a true Nash equilibrium solution for Texas Hold'em in the foreseeable future" makes the static strategy even more dangerous against a human player able to probe and test for specific weaknesses in his/her opponent's game.

Vexbot presented in Billings et al. [6] is an adjustment made from *PsOpti*, which replaces the supposedly optimal approach of finding minimax solutions with two variations to the Expectimax search, named miximax and miximix. In standard game theory, strategies usually revolve around three approaches: minimax, maximin, and maximax. A minimax strategy is one in which the goal is to minimize the maximal loss. This is as opposed to the maximin strategy which seeks to maximize the minimal gain. A maximax strategy, or greedy strategy, seeks to maximize the maximal gain. The three strategies return a singular action as the preferred action once the strategy resolves itself, but an Expectimax strategy, which seeks to maximizes the expected gain of a state from a pool of actions, considers all actions as possible for the given turn. From the Expectimax strategies, a miximax strategy does not preclude any option that is available to the opponent but instead applies a distribution over all possible actions by the opponent then the algorithm selects the action that returns the highest expected gain based on those distributions in future states. A miximix strategy is similar to the miximax in that all of the opponent's available actions are considered but instead of only selecting one, the presumed optimal action, the algorithm uses a probabilistic distribution to select an action. In the game tree search in the Texas Hold'em poker environment, the nodes representing opponent decisions are no longer modeled as selecting the child with the best return descending from each type of action the opponent may select, which in this case is from the set of check/call, bet/raise, fold, but a probability distribution of each action being selected based on the situation of the game and the opposing player strategy. Given that there are only three possible actions, the program can either elect to choose the action with the best possible return, resulting in the miximax solution, or apply a mixed strategy of its own, resulting in the miximix solution.

In the *Vexbot* program, there are three players modeled in the game tree: the two players involved in the game, and a neutral player whose responsibility is to reveal the public cards. Therefore, this results in four different types of nodes in the search tree: chance nodes, which represents the dealer's action, opponent decision nodes, which models the opponent's possible actions, player decision nodes, which models the player's possible actions, and leaf nodes, which represents the end of the current game. Each of these nodes possesses an expected value function calculating its

effect on the game. Equation 1 represents the expected value function for chance nodes. The goal of the formula is to calculate the effect that the public board cards being revealed has on the expected value on the rest of the game. $Pr(C_i)$ is the probability that public card i is revealed by the dealer amongst the possible n collections while $EV(C_i)$ is the expected value of the subtree descending from the revelation of the public card i.

$$\mathrm{E(C)} = \sum_{1 \le i \le n} \mathrm{Pr(C_i)} \times \mathrm{E(C_i)} \quad (1)$$

Equation 2 represents the expected value function for opponent decision nodes, like the chance node calculation, $Pr(O_i)$ is the probability that event $i \in \{f(old), c(all), r(aise)\}$ will be selected by the opponent and $EV(O_i)$ is the expected value of the subtree descending from each of those types of action.

$$\mathrm{E(O)} = \sum_{i \in \{f,c,r\}} \mathrm{Pr(O_i)} \times \mathrm{E(O_i)} \quad (2)$$

While O represents actions by the opponent, U represents actions by the player. Combined with the three possible actions, U_f is defined as the player opting to fold in the given turn, U_c is defined as the player opting to call, and U_r is defined as the player opting to raise. With these definitions, the expected gain of the player U, $E(U)$, is given as the following:

$$\mathrm{E(U)} = \max\left(\mathrm{E(U_f)}, \mathrm{E(U_c)}, \mathrm{E(U_r)}\right) \quad (3)$$

Equation 3 represents the player/program decision node for the miximax strategy where the expected value of the node is the maximal amongst the expected value of the subtree descending from each type of action. For a miximix strategy, a formula like Equation 1 and Equation 2 is used instead.

Equation 4 represents the expected value formula of the leaf node L. P_{win} is the probability of winning the hand (0 if player folds, 1 if opponent folds), $L_{\$pot}$ is the current size of the pot, and $L_{\$cost}$ is the amount the player paid into the pot.

$$\mathrm{E(L)} = \left(\mathrm{P_{win}} \times \mathrm{L_{\$pot}}\right) - \mathrm{L_{\$cost}} \quad (4)$$

While $\left(P_{win} \times L_{\$pot}\right)$ is typically the expected gain function on its own, the expected gain function $E(L)$ also takes into consideration the cost for the player to reach this terminal state in $L_{\$cost}$. The expected values from leaf nodes propagates back up the tree through the other three types of nodes to the point where the program is expected to decide based on the strategy employed.

To effectively model the opponent's behavior, a hierarchical structure with different levels of abstraction of behavior is constructed. The motivation behind this approach is the sheer amount of potential combination of cards, even when hands with similar strength are grouped together. This introduces the problem where certain groupings will not be observed when the situation demands it. The author suggests that the lowest level of abstraction is the actual betting pattern according to the strength of the hand, or the hand rank, of the opponent. Given that the number of times in which a player may make a raise call is limited during each betting phase, the author suggests that the next levels of abstraction should simply reflect the number of times the two players make such a decision regardless of who initiated the raise sequence. Once the different decision tree abstractions are constructed, a weight is given to each tree and the final opponent modeling decision comes from a combination of the trees dependent on the weight assigned. In addition, given that players tend to adjust their strategy during game play, older information is not as valuable as recent information. Therefore, a history decay factor is added to allow for a higher contribution by the more recent events to the calculation. Although the rate in which a human player may alter his/her strategy varies, the decay factor is a static value. If the strategy has not changed, adjusting the weight of older information does not alter the result of the

computation. If the strategy has changed, then the decay factor has accomplished its goal of reducing the weight of older and currently inaccurate information.

3. SIMULATION-BASED APPROACH TO GENERAL GAME PLAYING

The technique described by Finnsson and Björnsson [7] is applied to programs submitted for the annual AAAI General Game Playing competition. The goal of the competition is to create AI agents that can play a wide range of games based upon rules presented at run time by the competition. The rules are written in a specific language, called Game Description Language (GDL), which can be found in [8].

This approach was first proposed by Finnsson in [9] and subsequently expanded upon by Finnsson and Björnsson in [7, 10, 11]. In the annual general game playing competition held at the AAAI conference, CADIA-player was victorious in the years 2007, 2008, and 2012 (http://ggp.stanford.edu/iggpc/winners.php). It is important to note that unlike other papers discussed in this section, the games involved in the competition are perfect information games, or games where the entire state of the game is known to all involved players. Whereas games that are the focus of this dissertation involve imperfect information games, where the only knowledge the player possesses are those belonging to the characters the players control as well as the result of the interactions between those characters and the game world.

Given that the programmers do not have any information regarding the rules and goals of the games involved in each competition until the start of the games, the AI agents must be capable of extracting such information based upon the data provided when the competition begins. Usual techniques used by the more successful agents in the competition, as described by Finnsson and Björnsson in [7], involves applying a very small set of generic features in a game tree search algorithm with automatically learned heuristic evaluation function for pruning the branches and limiting the branching factor to lower the amount of processing time necessary. However, due to the wide variety of games that can potentially be presented to the agents, this technique runs the risk of utilizing features that will not reflect the key aspects of the game, resulting in very poor performance in the specific game.

The solution proposed by the author is to apply the idea of the Upper Confidence-bounds applied to Tree presented in [12] (UCT for the remainder of the paper) in conjunction with Monte Carlo simulation to evaluate payouts. The algorithm simulates the complete games repeatedly and collects information regarding the payout for different actions at each state and at the end of the deliberation period select the action with the best average expected gain. The agent using this technique, named *CadiaPlayer* by the author, will be able to avoid the use of the evaluation function for feature selection by examining all legal actions and thus the issue stated above will not affect the performance of this agent.

At each step in a Monte Carlo simulation, a number of trials are conducted with randomly selected actions. The results of the trials are collected, and the action selected at that step is the one with the highest level of average return value. The UCT algorithm introduces an extra term, named the UCT bonus, to balance the cost of examining untried paths (exploration) and the effect of exploring the same previously successful moves repeatedly (exploitation).

Let $A(s)$ be the set of all possible actions in the state s, $Q(s,a)$ is the average return value of action a in state s. The first part of the equation is in essence the Monte Carlo simulation action selection function. The additional term is the UCT bonus as stated above. $N(s)$ is the number of times in which the state s is visited while $N(s,a)$ is the number of times in which action a is selected at state s. The action selection algorithm is given by the following formula:

$$a^* = argmax_{a \in A(s)} \left(Q(s,a) + C \sqrt{\frac{\ln N(s)}{N(s,a)}} \right) \quad (5)$$

Performing action a at state s will increase both the numerator and the denominator of the UCT bonus. However, since the numerator includes a natural log, the rate of increase is lower than that of the denominator. In other words, performing action a at state s will decrease the UCT bonus for action a in the next instance state s occurs. Performing action a will also have the effect of increasing the UCT bonus value for all other possible actions at state s since the denominator will remain the same for those actions, but the numerator is increased with the appearance of state s. If an action a has not been selected previously, the algorithm defaults to selecting such an action for the next step of the simulation in order to create a point of reference. The aggressiveness in which the algorithm attempts to pursue suboptimal paths depends on the tuning variable C. The UCT term permits the algorithm to explore suboptimal steps in search of potentially better pay off in the future.

In addition to the function for determining next action in cases in which the average consequence of action a is known, as stated before, the algorithm defaults to selecting a previously unexplored action if such exists at a particular state. However, if there are multiple such actions available, an additional function based on the Boltzmann distribution first developed in the area of thermodynamics is used to determine which of the unexplored actions to test first. Given $P(a)$ is the probability of action a being chosen, $Q_h(a)$ is the average return of action a regardless of the state in which it is played, and τ is the tuning variable, the equation is given as the following:

$$P(a) = \frac{e^{Q_h(a)/\tau}}{\sum_{b=1}^{n} e^{Q_h(b)/\tau}} \quad (6)$$

A τ value approaching 0 will reduce the distribution to a maximax, or greedy, function while a high τ value will reduce the gap between the probabilities for the various options. The idea behind this function based on the observations from previous actions is that if an action is preferred in one state, it is likely to be preferred in another state even if the state-action pair has not been tested before in that specific state.

The variance between the different versions of *CadiaPlayer* lies in the process in which the simulation is conducted. Part of exploring previously unobserved state-action pairs involves comparing previously observed results from list of potential actions regardless of the state associated with those actions. In other words, the history of observed consequences of previous actions factors into the decision-making process at the current state. Finnsson and Björnsson [9, 7] introduce the technique named Move-Average Sampling Technique (MAST), which compiles the action score as the average of all occurrences of the given actions. This average is given as $Q_h(a)$ value in the previous equation. Finnsson and Björnsson [10] introduce the additional step named Rapid Action Value Estimation (RAVE), which maintains a separate record of values $Q_{RAVE}(s,a)$ in addition to the $Q(s,a)$ values, with the RAVE term being a state-action score value observed further down the tree. These RAVE terms are then used in combination with the observed $Q(s,a)$ values during the Monte Carlo process in instances where the state-action pair (s,a) is seen. This is accomplished by changing the $Q(s,a)$ term in the action selection function to $\beta(s) \times Q_{RAVE}(s,a) + \big(1-\beta(s)\big) \times Q(s,a)$ where $\beta(s) = \sqrt{\frac{k}{3\times N(s)\times k}}$ with $N(s)$ being the number of times state s is seen.

Based on this additional step, the denominator in the $\beta(s)$ term increases each time state s is observed and thus the weight of the RAVE term decreases each time state s is observed. This adjustment is based on the assumption that due to sample size, the value of $Q(s,a)$ is unreliable at the beginning of the Monte Carlo process, but the reliability increases as the number of observations increases. Lastly, Finnsson and Björnsson [11] introduce Features-to-Action Sampling Technique (FAST). The previous techniques give no consideration to the type of game being played or the rules of the game. FAST introduces two new features called piece type and cell location. The features maintain a list of values for each object where actions can be applied (piece type) as well as the result of the action (cell location). These values can be described as

contribution these piece types and cell locations made toward the accomplishing the goal of the game. This feature allows *CadiaPlayer* to select actions which provides greater contributes to the goals of the game with greater probability.

While the focus for Finnsson and Björnsson in [7, 10, 11] is the UCT with Monte Carlo search, since the General Game Playing Competition features both single-player and multiplayer games, Finnsson and Björnsson in [9] made note of the fact that *CadiaPlayer* is mainly designed for an adversarial game where there is at least one opponent. To handle situations where the system is presented with a single player game, *CadiaPlayer* makes use of the enhanced iterative-deepening A* search. For multiple player games however, in order to model opponent behavior, an UCT tree is constructed for each player involved in the game. The collection of UCT trees is then used in conjunction in order to project the most valuable next move. Note that the action selected by the previous two functions are only used in the Monte Carlo simulation phase, the resulting action selection is only dependent on the $Q(s,a)$ score from the simulation and the UCT bonus term is not involved. Although UCT trees are constructed for every agent involved in the game and the projection utilizes all the UCT trees, the author makes note of the fact that *CadiaPlayer* only evaluates the effect of the projected strategy on all agents in a two-player game, while the algorithm will only consider the strategy's effectiveness on itself in a multi-player game regardless of its consequence on all other agents.

Beside *CadiaPlayer*, UCT with Monte Carlo search are also used in other general game playing research. In fact, Swiechowski and Tajmajer [13] made note of the fact that Monte Carlo Tree Search (MCTS) is the algorithm of choice for every winner of the general game playing competition since 2007. The fact that MCTS only requires the full list of the rules of the game to operate makes the algorithm not only appealing to those working in the area of general game playing, but also the complex game of Go, including the famous Go playing program AlphaGo [14].

4. OPPONENT MODELING IN REAL-TIME STRATEGY GAMES

The technique proposed by Schadd et al. [15] is applied to modeling of opponent behavior in real time strategy game (RTS game for the remainder of the paper). Specifically, the algorithm is tested with an open source RTS game named SPRING.

The main challenges surrounding opponent modeling in RTS games are twofold. First in most RTS systems, most of the map of the battlefield is concealed by the "fog of war". The only area visible to the player is only within a certain range of any unit or building constructed. This imperfect information setting makes modeling an opponent strategy impossible unless some form of contact has been made between the two military forces, either in defense against an invasion by the opponent or in spying mission against the opponent's base. The contact must also be initiated early enough for the player to have sufficient time to respond to the strategy. The second issue is related to the last sentence in that, given that the game is real time system, whatever information gathered must be processed at real time. Therefore, any opponent modeling and strategy formation algorithms designed for such task must be able to resolve the computation rapidly with limited processing power given that, as referenced in [16], most of the recent games dedicate most processing power toward graphics rendering.

Given that the algorithm, unlike the approach discussed in Section 3, is tested on a specific game, predefined game specific knowledge may be added to this technique to improve upon its performance. However, this algorithm is general enough that it may be adapted across games of this genre with minimal adjustment. The technique proposed in this paper involves two parts. The first is a classification problem which seeks to identify which specific strategy employed by the opponent. The second is the counter-strategy selection algorithm based upon the model selected in the first part.

The first part of the algorithm seeks to match the opponent with a particular strategy. As stated before, most of the map is covered by the "fog of war" and thus any information about the opponent must be gathered when contacts are made. Such contact, as the author stated, can only be made during attacks by either side. The amount of time the opposition spends on attacks is therefore used as an identifier to classify whether the opponent is utilizing an aggressive strategy or a defensive strategy. In addition to the generic overarching classifier, the author also adds a second level in a hierarchical classification approach. By observing the specific information obtained during an attack, the algorithm also classifies which military unit is the focus of the opponent if the opposition utilizes an aggressive strategy and which end-game goal is being applied by the opponent if the opposition utilizes a defensive strategy. Since such information, either military unit in the first case or construction patterns in the second, can be obtained during battle, this approach does not violate the laws of the game and apply information that are otherwise unknown to the player.

Each type of strategy applied by the opponent has its own strengths and weaknesses. The goal of the second path is to find a counter strategy which avoids the opponent's strategy's strength while attacking its weaknesses. In order to select from one of the predefined strategies, we define π_i as the reward received at time i and δ as the discount factor. The following formula is used to calculate the expected gain, E, of each strategy:

$$E = (1-\delta) \times \sum_{i=1}^{\infty} \pi_i \times \delta^{i-1} \quad (7)$$

The discount factor is added to adjust the weights to favor immediate rewards over future rewards, especially since one may adjust his/her own strategy if it is proven to be unsuccessful, rendering any future rewards in doubt. Once the expected reward is calculated for each strategy, the strategy with the highest return will be selected by the algorithm.

Since a player may adjust his/her own strategy as stated above, the confidence level of the projected opponent strategy must be constantly updated. Given that the only instances when the opponent's behavior can be observed is during an act of aggression from either side, the confidence level, C_s, is adjusted after every moment of contact. Let δ be the discount factor, π be the reward received at this contact event, x be the current observed event, and $\psi_{s,t}$ be the probability factor representing the believe that a player is utilizing strategy s at time t, and is a representation of the information received during a contact event. The confidence level is then defined through the following formula:

$$C_s = \sum_{t=x}^{0} \psi_{s,t} \times \pi \times \delta^{x-t} \quad (8)$$

In discussing the technique, the author provides two additional points of interest. Information gathering, as stated throughout the paper surveyed, is one of the most important aspects in this algorithm and in general in a RTS game. However, how the information is obtained can vary greatly dependent on the approach of the opposition. Against an aggressive player, the information often presents itself during attacks by the opponent and one only needs to observe the type of units deployed for the attack to classify which specific strategy the opponent is utilizing. However, against a defensive player, the opponent often will avoid attacking until the end-game goal is near completion. In such instance, the player is responsible for securing information by deploying units for scouting mission. Therefore, an addition to the approach should feature the utilization of scouting mission dependent on the actual need for information on the opponent. Also, an aggressive strategy may be altered relatively easily through focus on construction of a different unit then before. However, a defensive strategy usually requires extensive time dedicated to the specific approach and cannot be changed quickly. Therefore, Schadd et al. [15] suggest that the discount factor applied in the two formulas should be dependent on this additional game specific information rather than a "one-size-fits-all" constant factor.

While the initial work limits the observations to the current game and makes decisions based solely on those observations and expert knowledge relating to the game environment. Subsequent works by the same group in [17, 18, 19] introduce an addition level of opponent modeling by applying case-based reasoning. Case-based reasoning, in simple terms, is the approach to solve the current problem based on experiences in attempting to solve previous, similar problems. While strategy games require a certain amount of time to complete, each run of the game is independent from each other and there is no carry over bonus or penalty based on the result of the previous game. Therefore, the approaches presented in [17, 18, 19] make use of recorded information of previous runs of the same strategy game and cluster the opponent strategies into groups in the offline learning phase of the algorithm. During the game, the online application of the algorithm matches the current knowledge of the game world, particular any information that can be gathered regarding the opponent, with the various clusters to discover the subset of previous playthroughs that most resemble the current situation. Afterward, the algorithm selects from those playthroughs the strategy that best resembles the goal of the current game. This approach allows the algorithm to firstly respond quickly to changing situations in the game world, and secondly perform at a level that is not the absolute optimal solution. In other words, this approach can also function as a difficulty adjustment algorithm instead of simply as an optimal performance algorithm.

5. CREATING PRO-LEVEL AI FOR A REAL-TIME FIGHTING GAME USING DEEP REINFORCEMENT LEARNING

Oh et al. in [20] focuses the research on the fighting game genre. The algorithm is tested in the game Blade & Soul (NCSoft 2012). In particular, its two-player dueling system called B&S Arena Battles available in the game.

The authors open the paper by presenting the challenges the developed system must overcome.

The first is, as with most modern games, the size of the range of possible actions at each time step. In the case of this paper, given that the game utilizes a real-time system, the authors define a time step as one-tenth of a second in real time. Within each time step, the player must decide on all three components which make up the action decision: the direction in which the character will move toward; the direction in which the action will be directed toward, and the actual action that will be performed by the character. Therefore, the number of possible options at each time step is the product of the size of the range of each component. For the purpose of the paper, the authors estimate that number to be 144. The second issue is connected to the first in those certain actions can only be performed in certain specific states. In other words, the availability of certain actions at a particular time step may depend on what actions were chosen during the previous time steps. Lastly, also as with most modern games, the game operates as an imperfect information system. In other words, the players have no knowledge regarding the other player's decision-making process.

Given both the real-time nature and the imperfect information environment, the authors describe the game as "a series of rock-paper-scissors games" and that "the essence of the problem is to approximate a Nash equilibrium strategy so that the agent can respond appropriately to any opposing strategy." In other words, the algorithm will give no consideration to modeling the behavior of the opponent and will simply focus its effort in creating the optimal version of the agent, or the character controlled by the algorithm, which is reasonable in this case given the limited amount of time available for processing for each action decision (one-tenth of a second in this case).

To that end, the authors define a Markov decision process (MDP) $= \{S, A, P, R, \gamma\}$ where S is the set of all states, A is the set of all actions, P is the state transition probability defined as the probability that the game will reach a particular state given the current state and an action, R is

the reward or gain of the action by a character at a given time, and γ is the discount factor which determines the weight of immediate rewards versus long term rewards [21, 22].

In addition, the authors define a policy $\pi: S \rightarrow A$, which defines the action performed given a specific state. The policy is given as $\pi(s_t) = a_t$, defined simply as perform action a at time t, defined as a_t, given state s at time t, defined as s_t. Define the reward at time t, or the reward of transitioning from state s_t to state s_{t+1} as a result of performing action a_t, as r_t and the discount factor at time t as γ_t. Given that the transition from s_t to s_{t+1} is locked in by the definition of rewards, the probability of transitioning from s_t to s_{t+1} no longer factors into the expected gain equation. The optimal policy π^* is therefore defined by the following equation:

$$\pi^* = argmax_\pi E_\pi(\sum_{t=0}^{\infty} r_t \times \gamma_t) \quad (9)$$

Furthermore, as stated previously, the strategy of the opponent's decision-making process is unknown to the algorithm, the authors make a further assumption that the pool of policies for the opponent is fixed and thus the policy of the opponent can be expressed as a function of the action of the agent. Given such an assumption, the MDP can be modified such that only the actions of the agent need be considered.

Given the nature of fighting games, the reward of selected actions is computed by the health point information (HP). For this algorithm, at the terminal state of each battle, defined as one character having zero remaining life points, the reward rwint is given as +10 for a win and −10 for a loss. For non-terminal states, $r_t^{win} = 0$. In addition, for all states, there is an additional reward determined by health of the two characters given as r_t^{HP} and is computed as the difference of the difference in health point of the agent $\left(HP_t^{agent}\right)$ and the difference in health point of the opponent $\left(HP_t^{opponent}\right)$ in the current time step versus the immediate previous time step $\left(HP_{t-1}^{agent}\right)$ and $\left(HP_{t-1}^{opponent}\right)$.

$$r_t^{HP} = \left(\mathrm{HP}_{\mathrm{t}}^{\mathrm{agent}} - \mathrm{HP}_{\mathrm{t-1}}^{\mathrm{agent}}\right) - \left(\mathrm{HP}_{\mathrm{t}}^{\mathrm{opponent}} - \mathrm{HP}_{\mathrm{t-1}}^{\mathrm{opponent}}\right) \quad (10)$$

The base equation for r_t is thus defined as the following:

$$r_t = r_t^{win} + r_t^{HP} \quad (11)$$

Note that, as stated in the title of the paper, the algorithm is based on a deep reinforcement learning technique. Deep reinforcement learning (Deep RL) is a combination of two separate techniques: deep learning and reinforcement learning. A reinforcement learning system is one in which autonomous agents learn how to best perform a given task by exploring possible actions through a series of trial-and-error processes. This system is typically defined in the form of a MDP like one defined above. A deep learning system is a system of learning based on artificial neural networks (ANN), in particular deep neural networks (DNN), which is an ANN with many layers.

For the algorithm presented by Oh et al. [20], rather than creating only a single type of agent as well a single type of opponent for the agent to train against, the authors created three types of play styles by introducing additional factors to the rewards formula. The first is a time penalty, which penalizes the character for prolonging the battle. In other words, the longer the battle proceeds, the less rewarding the actions become. The second is a distance penalty, whereby the greater the physical distance between the agent and the opponent, the less rewarding the actions become. The last is a health ratio, which are weight factors added to both the agent and the opponent contribution in the r_t^{HP} formula.

With these three additional factors in the calculation of the reward formula, the authors create three different play style. The aggressive play style seeks to continuously press forward against the opposition and inflict as much damage to the opposition as quickly as possible. Given these properties, the authors define the aggressive play style by assigning a comparatively higher time

penalty (0.008 in the experiment), a comparatively higher distance penalty (0.002 in the experiment) and maintains a balanced health point weight ratio (50%:50% in the experiment). The defensive play style seeks to preserve the character's health as much as possible even if it means maintaining a certain distance from the opposition to avoid suffering any damage and prolonging the battle. The defensive play style is therefore defined by no time or distance penalty while giving a higher health point weight ratio to the agent's health point versus the opponent's health point (60%:40% in the experiment). The third play style is termed a balanced play style and is in between the aggressive play style and the defensive play style. This play style is still given a time penalty (0.004 in the experiment) and a distance penalty (0.0002 in the experiment), but both penalties are less than the aggressive play style. Like the aggressive play style, the balanced play style is also given an equally distributed health point weight ratio (50%:50% in the experiment).

As stated in the beginning of this section, the time steps are partitioned into one-tenth of a second intervals. It would be impractical to expect a constant stream of action during each time step. To that end, the authors devise two data skipping techniques to discard certain information during the self-play process in training the agents. Recall that the action decisions are partitioned into three components: the specific action to be selected, the direction for the character to move in, and the direction in which the action will be aimed. In fighting games, the movement direction and the action direction typically align with each other. In other words, the actions are typically aimed at the direction in which the character is moving in. Therefore, the action decision is only defined by the action selected and the direction selected and the two data skipping techniques focus on these two aspects of the action decision.

Regarding action selection, the authors define the "no-op" action which signifies the time steps where no action is selected. These "no-op" actions are divided into passive "no-op" in which the game state dictates that the agent cannot perform any action, and active "no-op" in which the agent makes the decision to not perform any action during the current time step. While active "no-op" reflects the strategy of the agent, passive "no-op" does nothing to contribution to strategy formulation as its use is enforced by the environment and not the agent. Therefore, passive "no-op" are discarded by the algorithm in both the training of the agents and the evaluation of the rewards.

In terms of movement, the authors make the argument that the distance traveled by the character during a given time step is very limited because the time step is contained within such a brief moment in time. It is impractical for a character to repeatedly move in different directions during consecutive time steps. Therefore, the agents are forced to maintain the move direction for a predefined number of time steps. In other words, once a movement direction is selected, policies which involve moving in a different direction is ignored for a predefined duration. This will allow the move action to reflect real in-game actions rather than simply performing a random walk across the game area.

As stated, the algorithm is a deep reinforcement learning system, and the agents are trained through a self-play process. The experiment to test the learning algorithm is partitioned into three parts. The first experiment tests the effect of creating the three different play styles. The agents with the three play styles are trained against a shared pool of past selves from all three play styles, while the baseline agent is assigned the base reward equation and is trained against of pool of only its past selves. After the agents are trained, the agents with modified play styles are tested against the trained baseline agent. The result of this experiment shows that the agents with modified play styles have a winning percentage of approximately 60% and thus it can be concluded that the use of various play styles will provide a better result than simply using the baseline agent.

While the first experiment shows that agents with various play styles are beneficial, a key component of the experiment set up is the shared pool of past selves that the agents are trained

against. With the shared pool, each agent can potentially face a copy of a past self of a different play style. The second experiment tests the performance of agents training against the shared pool versus agents training against independent opponent pools where agents can only face against a copy of a past self with the same play style as the agent in training. The assumption of the authors is that agents trained with the shared pool will perform better against opponents it has never encountered. Therefore, each trained agent will be tested against the two agents with different play styles that are trained with an independent opponent pool. The result of the experiment supports the assumption that the agents trained with the shared pool outperform the agents trained with independent pools. The last experiment involves testing the trained agents against professional human players. The result of the matches against professional players shows that the AI agents trained using this method can perform at minimum on par with professional championship level players, thus showcasing the effectiveness of the algorithm.

6. ROLLING HORIZON EVOLUTIONARY ALGORITHMS FOR GENERAL VIDEO GAME PLAYING

Gaina et al. [23] seek to optimize the Rolling Horizon Evolutionary Algorithm (RHEA) using N-Tuple Bandit based approach. The optimization method operates by combining various approaches drawn from multiple literature as well as new modifications introduced in this paper. The full algorithm is then tested against 20 games from the General Video Game AI Framework with varying results.

The central concept of the algorithm revolves around the evolutionary algorithm. Given the large state space of the games where the full algorithm is tested on, the author opts for the use of RHEA, which is an evolutionary algorithm where the length of the action sequences evaluated by the algorithm is bounded rather than seeking the full solution. Outside of the size limitation, RHEA operates in the exact same manner as any typical evolutionary algorithm. The algorithm begins by randomly creating a collection of action sequences. A heuristic function is defined to evaluate the game state at the conclusion of each action sequence to produce the fitness or desirability of the given action sequence. The best performing sequences are kept within the collection and additional sequences are created by combining two sequences in the previous iteration of the collection. Those newly created sequences are then mutated before being added to the collection for the next round of evaluation. The process repeats until a predefined termination points typically in the form of either a generation limit or a time limit.

The authors then modify the standard RHEA with the N-Tuple Bandit method to form the N-Tuple Bandit Evolutionary Algorithm (NTBEA) which is the focus of the paper. Let L be the size limit of the action sequences, NTBEA creates a collection of n-tuples where the terms of the lists are drawn from the list $[0, L-1]$ without replacement. Essentially, the collection of n-tuples is a collection of index sets of the action sequence. For a given action sequence, a set of neighbors is generated through uniform mutation of the input sequence. Once the collection of index sets and the neighbors of action sequences are generated, the following values are collected for each of the n-tuples:

Let m be the total number of n-tuples, T_j be the jth n-tuple in the collection, o be an action sequence, $Q(T_j)$ is the average fitness of the neighbors, $N(T_j)$ is the number of times T_j was sampled, and $N(T_j, o)$ be the number of times T_j was sample and the values of the neighbor at the indices defined in T_j is the same as input sequence o. The evaluation of action sequence o is then given by the following upper confidence bound (UCB) equation:

$$\mathrm{UCB_o} = \frac{1}{\mathrm{m}} \sum_{\mathrm{j}=1}^{\mathrm{M}} \left(\mathrm{Q(T_j)} + \sqrt{\frac{2\,ln\,\mathrm{N(T_j)}}{\mathrm{N(T_j,o)}}} \right) + \mathrm{noise} \quad (12)$$

The random noise parameter is added to the equation to randomly break ties.

The process then repeats with the neighbor of o with the highest UCB value serving as the input action sequence from which a new set of neighbors is generated. The algorithm terminates after a given number of iterations or a time limit is reached and the action sequence with the highest average fitness value is selected for the current step. The N-Tuple Bandit approach is done as a tuning method to the RHEA to ensure that the entire population consists of, at minimum, points of local maxima to improve the performance of the evolutionary algorithm.

In addition to the use of N-Tuple Bandit, the paper introduces several parameters which further refines the overall algorithm. These parameters are partitioned into two subsets: the first governing structure of the evolutionary algorithm, and the second controls how the evolutionary algorithm is applied during game play.

The concepts of selection, crossover, and mutation form the basis of evolutionary algorithm and this paper introduces various means in which these three concepts are applied. In terms of selection, which governs which action sequences are used to produce the next generation, the options are tournament, roulette, and rank. The tournament approach is defined as randomly selecting a subset of the population then choosing those sequences with the best fits to produce the next generation. The roulette approach is defined as assigning probability to the sequences in accordance with their fitness value then using those probability to randomly select the parents. The rank approach assigns ranks to the sequence with those with the lowest fitness values ranked first. The sequences are then assigned probabilities equal to their rank in the parent selection process. The means in which the selected parents are then used to produce the next generation is then control by the concept of crossover. In terms of the approach presented in this paper, there are a total of two types of crossovers: uniform and n-point. The uniform approach creates the new sequence by randomly selecting individual actions from either parent with equal probability to fill the corresponding spot in the new sequence. The n-point approach partitions the two parent sequences into a collection of size n sub-sequences before alternatively selecting sub-sequences to form the new action sequence. The last concept, mutation, governs how the newly created action sequences are altered before being added to the new population. The mutation steps are partitioned into four different approaches: uniform, softmax, diversity, and n-bits. Uniform mutation alters each step within the sequence with equal probability and assigned a new value with equal probability. Softmax mutation uses the softmax equation [24] to governs the likelihood of mutation. Diversity mutation focuses the mutation to steps with the least amount of information available and sets its value to the value with the least amount of information available. N-bit mutation selects a sub-sequence of n steps and mutate the selected sub-sequence to different values.

Beyond the above concepts, there are two more parameters defining the structure of an evolutionary algorithm. The first parameter controls the evaluation process of the action sequences by defining what constitutes the heuristic function. The second parameter controls the starting state of the evolutionary algorithm by governing how the initial population is generated. In terms of the heuristic function, the authors propose several evaluation methods to gauge the effectiveness of an action sequence. Each version has its own meaning in terms of what the heuristic function is measuring. The examples given by Gaina et al. [23] are the following: $F[L-1]$, or only maintaining the value of the terminal state of the sequence; $\Delta(F[L-1], F[0])$, or the difference between the value of the starting state and the terminal state; $\bar{F}$, or the mean of the values of each individual state along the action sequence; $min(F)$, or the minimum value along the action sequence; $max(F)$, or the maximum value along the action sequence; and finally $\sum_{i=0}^{L} F[i] \times \gamma^i$, a weight sum of all the values along the action sequence. In terms of the initialization of the RHEA, the standard approach involves simply randomly generating the initial population. The author introduces two additional initialization process. The first is the one step look ahead method (1SLA), which evaluates all possible actions in the initial state and select the

action with the highest value given the heuristic function. The process is repeated with the next state generated by the selected action until the targeted length is reach. If the end state is reached prior to the targeted length, the sequences is padded with random actions until the targeted length is reached. The remaining population pool is then generated by mutating the initial selected sequence. The second method utilizes a depth limited Monte Carlo tree search method (MCTS) to find the optimal initial sub-sequence before padding the remainder of the sequence with randomly selected actions. The remainder of the population pool is generated by mutating the initial action sequence.

The parameters governing how the RHEA is applied mainly revolves around four points: how often the algorithm is run during the game play, what information is retained between runs of the algorithm, determining how far ahead the evolutionary algorithm should evaluate, and how much additional information should be gathered during the simulation phase of the RHEA. The first point is governed by what the authors term "frame skipping". The idea behind frame skipping is the desire to seek a balance between response time and the depth of the evaluation process. If frame skipping is not applied, then a decision is made at every game step. This approach allows for a rapid response to sudden shifts in the game world at the cost of a very limited time frame for evaluation. If frame skipping is applied, then the algorithm can extend the evaluation across multiple game steps which in theory should result in better decision making. This benefit comes at a cost to the AI being restricted to pre-defined actions during game steps where no decisions are made. The authors offer four different types of pre-defined actions for those game steps: repeat, which performs the same action over and over until the next decision is made; null, which performs no actions until the next decision is made; random, which randomly selects actions until the next decision is made; and sequence, which continues with the action sequence selected rather than simply playing the first action and abandoning the remaining sequence. In terms of information retention, the idea behind this concept is tied to the previous point regarding evaluation time. The author defines this approach, name "shift buffer", as carrying the final population of the RHEA from the previous game step as the initial population for the RHEA during the current game step with the modification that the action representing the previous game state be removed and the sequences padded with a new random action to form sequences of the desired length. Regarding how far ahead the RHEA should evaluate, the authors suggest the use of "dynamic depth". Recall that the target of the algorithm is general game playing, which involves games across a multitude of genres. Each type of games has its own rewards method which the authors categorize into no rewards, which only grant rewards during end game conditions; dense rewards, which provides steady feed backs on the results of actions; and discontinuous rewards, which only provides intermittent feed backs. During periods of games where reward information is provided regularly, it may behoove the system to shorten the look ahead period to better attune to any potential shifts. During periods of games where reward information is less frequent, the system may consider lengthening the look ahead period to better gauge which actions are beneficial. Finally, with regards to simulation, the authors suggest the additional of a limited level of Monte Carlo rollouts to extend the evaluation of action sequences to pad the evaluation process with the addition of a Monte Carlo method.

7. CONCLUSION

We have reviewed papers from the game AI area across two time periods. The first set of papers, each drawn from the late 2000s to early 2010s, mainly approach the issue of game AI using tree search. These approaches focus on developing a model of player behavior and representing this model in the form of tree structures then performing the search on these constructed models. The second set of papers, drawn from the beginning of the 2020s to current time, approach the development of game AI using much more complex algorithms like neural network and evolutionary algorithm. While these methods were theorized very early on in studies of machine learning techniques, meaningful research utilizing these techniques were not popularized until the

turn of the century when computer hardware capacity has increased to the point where the use of these techniques become viable. Given that application of these techniques on specific topics are not typically studied until a certain level of understanding into their inner workings have already been reached, it is understandable that the deployment of these techniques in game AI is not a focus of researchers until the late 2010s into the early 2020s. By the same reasoning, we can expect developments in game AI to continue to expand with newer machine learning technologies currently in development or even to be developed in the future.

REFERENCES

[1] Billings, D., D. Papp, J. Schaeffer, and D. Szafron (1998b). Poker as a testbed for ai research. In Conference of the Canadian Society for Computational Studies of Intelligence, pp. 228–238. Springer.

[2] Billings, D., D. Papp, J. Schaeffer, and D. Szafron (1998a). Opponent modeling in poker. Aaai/iaai 493(499), 105.

[3] Davidson, A., D. Billings, J. Schaeffer, and D. Szafron (2000). Improved opponent modeling in poker. In International Conference on Artificial Intelligence, ICAI'00, pp. 1467–1473.

[4] Billings, D., A. Davidson, J. Schaeffer, and D. Szafron (2002). The challenge of poker. Artificial Intelligence 134(1-2), 201–240.

[5] Billings, D., N. Burch, A. Davidson, R. Holte, J. Schaeffer, T. Schauenberg, and D. Szafron (2003). Approximating game-theoretic optimal strategies for full-scale poker. In IJCAI, Volume 3, pp. 661.

[6] Billings, D., A. Davidson, T. Schauenberg, N. Burch, M. Bowling, R. Holte, J. Schaeffer, and D. Szafron (2004). Game-tree search with adaptation in stochastic imperfect-information games. In International Conference on Computers and Games, pp. 21–34. Springer.

[7] Finnsson, H. and Y. Björnsson (2008). Simulation-based approach to general game playing. In Aaai, Volume 8, pp. 259–264.

[8] Genesereth, M., N. Love, and B. Pell (2005). General game playing: Overview of the aaai competition. AI magazine 26(2), 62–62.

[9] Finnsson, H. (2007). Cadia-player: A general game playing agent. Ph. D. thesis, Citeseer.

[10] Björnsson, Y. and H. Finnsson (2009). Cadiaplayer: A simulation-based general game player. IEEE Transactions on Computational Intelligence and AI in Games 1(1), 4–15.

[11] Finnsson, H. and Y. Björnsson (2011). Cadiaplayer: Search-control techniques. KI-Künstliche Intelligenz 25, 9–16.

[12] Kocsis, L. and C. Szepesvári (2006). Bandit based monte-carlo planning. In European conference on machine learning, pp. 282–293. Springer.

[13] Swiechowski, M. and T. Tajmajer (2021). A practical solution to handling randomness and imperfect information in monte carlo tree search. In 2021 16th Conference on Computer Science and Intelligence Systems (FedCSIS), pp. 101–110. IEEE.

[14] Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. (2016). Mastering the game of go with deep neural networks and tree search. nature 529(7587), 484–489.

[15] Schadd, F., S. Bakkes, and P. Spronck (2007). Opponent modeling in real-time strategy games. In GAMEON, pp. 61–70. Citeseer.

[16] Machado, M. C., E. P. Fantini, and L. Chaimowicz (2011). Player modeling: Towards a common taxonomy. In 2011 16th international conference on computer games (CGAMES), pp. 50–57. IEEE.

[17] Bakkes, S., P. Spronck, and J. Van Den Herik (2008). Rapid adaptation of video game ai. In 2008 IEEE Symposium On Computational Intelligence and Games, pp. 79–86. IEEE.

[18] Bakkes, S., P. Spronck, and J. Van den Herik (2009). Rapid and reliable adaptation of video game ai. IEEE Transactions on Computational Intelligence and AI in Games 1(2), 93–104.

[19] Bakkes, S., P. Spronck, and J. Van Den Herik (2011). A cbr-inspired approach to rapid and reliable adaption of video game ai. In Case-Based Reasoning for Computer Games Workshop at the International Conference on Case-Based Reasoning (ICCBR), pp. 17–26. Citeseer.

[20] Oh, I., S. Rho, S. Moon, S. Son, H. Lee, and J. Chung (2021). Creating pro-level ai for a real-time fighting game using deep reinforcement learning. IEEE Transactions on Games 14(2), 212–220.

[21] Bellman, R. (1957). A markovian decision process. Journal of mathematics and mechanics, 679–684.

[22] Howard, R. A. (1960). Dynamic programming and markov processes.

[23] Gaina, R. D., S. Devlin, S. M. Lucas, and D. Perez-Liebana (2021). Rolling horizon evolutionary algorithms for general video game playing. IEEE Transactions on Games 14(2), 232–242.

[24] Franke, M. and J. Degen (2023). The softmax function: Properties, motivation, and interpretation.