The Best Offense: Game Theory in Video Game Enemies

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Bachelor of Arts in the
Departments of Computer Science and Mathematics at The
College of Wooster

by
Nicholas Czaban
The College of Wooster
2019

Advised by:

Dr. Denise Byrnes (Computer Science)

Dr. Pamela Pierce (Mathematics)

THE COLLEGE OF
WOOSTER

Abstract

As computer technology has improved, the complexity and variety in computer video games has evolved alongside. In particular, computer-controlled players in these games have grown significantly more intelligent over time. The decisions and strategies of these opponents can be studied and assessed with game theoretic techniques. This thesis demonstrates these techniques to develop strategies for the computer-controlled opponent in a role-playing video game.

Acknowledgments

I would like to thank my advisers, Dr. Pierce and Dr. Byrnes, for all their advice and encouragement on this thesis. I must also thank the students who participated in my survey and tested my game. Commendations go to all the professors, both at the College of Wooster and in my pre-college career, who helped foster my love of learning. Finally, my gratitude goes out to all the friends and family members who have supported me over the years.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

Introduction

As the computing power of personal computers has increased, the depth and variety of interactive computer games has improved at a commensurate rate. The earliest video games - *Spacewar!* (1962) and *Pong* (1972), for example - required two human players, and did not have any digital opponents. By the late 70's, games like *Pursuit* (1975), *Space Invaders* (1978), and *Asteroids* (1979) introduced computer-controlled opponents which moved in set patterns.

Figure 1.1: A screenshot from the game *Space Invaders*. The top line of enemy ships has reached the left side of the screen, so all enemy ships are in the midst of advancing forward [22].

In Space Invaders, enemies move in horizontal lines and occasionally shoot at the player, as shown in Figure 1.1. The player can move horizontally, to dodge projectiles and hide behind shields, and shoot back at the enemy. When a line of enemies hits a wall, the ships advance toward the player. As shown in Figure 1.1, the top line of enemy ships has reached the left wall and all ships are advancing one row forward. As more enemy ships are destroyed, the lines of enemy ships become faster and faster, making them harder to hit. If an enemy ship advances all the way to the player, the game ends and the player loses. Although the game becomes more difficult as the game progresses - ships become faster and shields become riddled with holes - this difficulty has little to do with decisions made by the enemy ships; they just continue moving across the screen and shooting wildly.

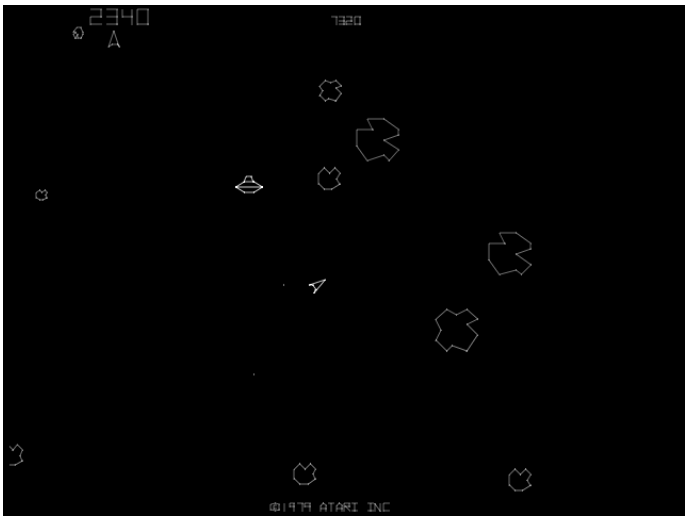

Figure 1.2: A screenshot from the game *Asteroids*. The player controls the triangular ship, and the flying saucer shoots in random directions. Another type of saucer exists in the game which fires directly at the player [4].

In Asteroids, the player's ship is freely movable in space. The ship must avoid or destroy the asteroids flying across the screen, and occasionally destroy two different types of flying saucers which appear from off-screen. One saucer, the larger of the two, fires shots randomly. The other, smaller saucer targets the player directly. In addition, the asteroids break into smaller asteroids as the player shoots them. Again,

the difficulty of the game increases as asteroids break into smaller, faster pieces, but this does not reflect any choices made by the asteroids. The asteroids themselves simply move as the law of inertia dictates. The first saucer, which shoot randomly, also does not make any choices which make the game more difficult. Only the second saucer, which directly targets the player, has a semblance of intelligence. We infer from the nature of the gameplay that the saucer wants to destroy the player. By firing right at the player, this enemy takes deliberate steps to achieve this goal. In contrast, the saucer which shoots randomly will sometimes try to achieve their goal (when the randomly-fired projectile moves in the general direction of the player) and sometimes will not (when the projectile moves in the complete opposite direction).

As time progressed, video game enemies were given more intricate patterns (as in the space shooter *Galaga* (1981)) and more heavily-scripted responses to inputs (as in the first-person shooter *Half-Life* (1998)) [13].

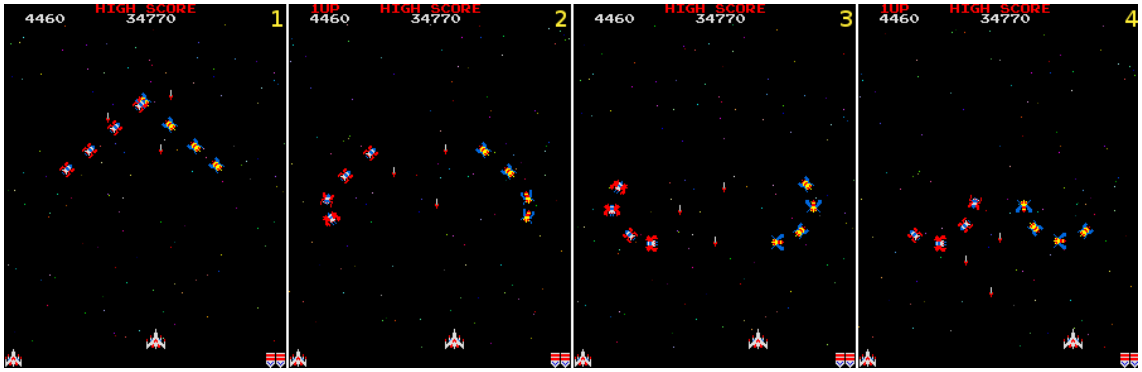

Figure 1.3: Four screenshots from the game *Galaga*, showcasing (from 1 to 4) the movement of enemy ships [8].

Galaga, for example, uses curves and loops in the flight paths of enemy spaceships, as can be seen in Figure 1.3. Compared to the straight lines of *Space Invaders*, the spaceships are more challenging to hit. Furthermore, the spaceships are programmed to fire when the player is directly below them, while *Space Invaders* has enemies shooting at random [13].

In more recent titles, AI systems begin to show more real intelligence. Some of this comes from more complex routines, as seen in the final boss of the game *Final Fantasy IX*.

```
1  set selectedattack = RandomAttack( attacklist )
   if ( selectedattack == Flare )
3  set SV_Target = RandomInTeam( NotMatching(SV_PlayerTeam[
     STATUS_CURRENT], PETRIFY | DEATH | ZOMBIE | REFLECT) &
     NotMatching(SV_PlayerTeam[STATUS_AUTO], REFLECT) )
   elseif ( selectedattack == Holy )
5  set SV_Target = RandomInTeam( NotMatching(SV_PlayerTeam[
     STATUS_CURRENT], PETRIFY | DEATH | ZOMBIE | REFLECT) &
     NotMatching(SV_PlayerTeam[STATUS_AUTO], REFLECT) )
   elseif ( selectedattack == Meteor )
7  set SV_Target = SV_PlayerTeam
```

Listing 1.1: Section of AI script for Necron, an enemy in the game *Final Fantasy IX* [1].

This boss, named Necron, has a variety of possible actions; three of these actions are shown in Listing 1.1. In line 1, a random type of attack is selected from a list of possible actions. For two of these attacks, in lines 3 and 5, one character in the player's team is randomly selected, provided that they do not have one of the listed special statuses. Two of these statuses (PETRIFY and DEATH) refer to when a character controlled by the player can no longer perform actions of their own [20]; PETRIFY turns a character to stone, and DEATH is self-evident. In this case, Necron targets a random character in the player's party who is not petrified or dead. The REFLECT status prevents a player character from being affected by magic spells; the spells are reflected back at the opponent instead [20]. For instance, if Necron did not exclude characters with the REFLECT status and tried to cast its Holy or Flare spell on a character with REFLECT, the spell would not damage said character.

Instead, Holy or Flare would be reflected and deal damage to Necron, effectively causing the boss to damage itself. Thus, by excluding characters with REFLECT from the list of possible targets, Necron avoids making costly mistakes in battle. These checks improve the perceived "intelligence" of the boss; the AI does not cause itself harm from reflected magic or waste actions to attack incapacitated players.

While the *Final Fantasy IX* example employs predetermined choices (apart from some random selection), other forms of intelligence come from more open-ended coding. For example, stealth games feature guards who must react to a variety of situations. One technique for such AI is to use fuzzy logic [13]. Fuzzy logic deals with sets where membership is on a continuum [24]; is is possible for an element to only be partially in a fuzzy set. In this case, a guard in a stealth game would use this type of logic to transition from a calm state to a suspicious state to an alerted state as the player's presence is detected [13]. Fuzzy logic is essential in stealth games to account for distance; if a player makes a noise two rooms away, the guard shouldn't have as strong a reaction as when the player makes a noise directly behind a guard. But, players may also make noise intentionally to distract guards. Thus, the AI for a guard cannot be too strongly scripted; what one player may do as a distraction another player may do accidentally, and the AI must account for these differences in play.

In many role-playing games (RPGs) like *Final Fantasy IX*, the heroes live in fantastical worlds and often have the ability to heal themselves of their wounds with quick use of a potion or magic spell. While the player-controlled characters have these abilities, their AI-controlled opponents frequently do not. In rare cases where enemies can heal themselves, this ability is often relegated to one-time uses determined by the game designers. For instance, an enemy may heal itself when it is close to death, but only the first time. When said enemy is once again close to death, they will not attempt to heal again. The goal of an opponent in a role-playing game

should logically be survival, yet their choice to only heal once does not support that goal.

In order to make an AI opponent in a role-playing game which displays intelligence, the opponent should not be unduly limited in its ability to heal. This thesis concerns itself with designing AI opponents that can decide when to use a healing ability, and using game theory to analyze the effectiveness of these AI's strategy.

CHAPTER 2

Extensive Form Games

2.1 Basics of Game Theory

Frequently used and originally explored in economic contexts, game theory is defined as the study of interactions between decision-makers. Antoine Cournot attempted to explain the decisions of duopolies (markets with only two sellers producing a product) in 1838 using the concept of strategic behavior [23]. In this case, strategic behavior refers to scenarios where one person's decisions both affect and are affected by the decisions of other persons. Cournot later influenced Joseph Bertrand's 1883 work, which examined strategic decisions in product pricing. These early efforts were limited by standard economic methodologies; it wasn't until John von Neumann's 1928 proof of the minimax theorem and his later collaboration with economist Oskar Morgenstern that the modern field of game theory was formed [23]. The strategies involved with game theory are used to explain the collusion and subsequent breakdown of oligopolies like OPEC, and the symbiosis between sharks and pilot fish.

As we begin our study of game theory, it is important to precisely define what games are.

Definition 1. *A game is a description of a strategic interaction which sets the interests of players and constrains the actions a player can perform, but does not describe specifically which actions are performed [9].*

Example 2.1.1. Chess fits the definition of a game. In chess, players have a set of pieces, each with their own type of movement and attack pattern. The most important piece is the King, which can move and capture one space in any direction. However, if the King is under threat (i.e. an opponent's piece can capture the King on the next turn), then the King is considered to be in "check." If the King is in check, cannot leave check, and no piece can capture or block the threatening piece, the King is in checkmate and the game ends.

The player's actions in chess are constrained by the rules of the game and how each individual piece can move. The interest of the player is to checkmate the other player's king. However, the game does not specify which moves are taken to place a king in checkmate.

Game theory involves the techniques used to analyze and determine solutions, strategies, and outcomes for various games [9].

Definition 2. *A solution of a game is the description of outcomes which may arise from the game [9].*

For the chess example, a single outcome of the game would be the sequence of actions which lead from the starting position of the game to a checkmate. The solution of chess is thus the collection of all sequences going from start to checkmate.

There are several types of solutions used in game theory, each being more useful in some games than in others [9]. Three solutions - Nash equilibrium, mixed strategies, and subgame equilibria - will be discussed later in this chapter.

It is assumed that players in these games are both *rational* - they pursue well-defined objectives and act in their own self-interest - and *strategic* - they take the

knowledge and behavior of other decision-makers into account [9]. With these assumptions, each decision made by a player is deliberate and involves optimization of some sort. Players consider their actions and the consequences thereof, and choose actions that best fit their preferences [9]. Returning to the chess example, players pursue their goal of checkmating the opponent's King, and they must plan ahead and predict what their opponent might do in defense.

The preferences of these players are represented using a utility function. In game theory, *utility* is the quantitative measure of a choice's value; depending on the structure of the game, this can be a monetary value or some expression in terms of the happiness a player gains from that outcome. A highly desired outcome for a player will have a higher utility than an undesirable outcome. In many games, the desired outcome for a player stands in diametric opposition to the desires of other players. These are referred to as *strictly competitive games* [9]. One type of strictly competitive game is a *zero-sum game*.

Definition 3. *A game is "zero-sum" if an action made by one player increases their utility by an amount equal to the loss in utility for the second player from that same action. Specifically, for an action a and utility functions u_1 for player 1 and u_2 for player 2, $u_1(a) + u_2(a) = 0$. Zero-sum games can only have two players [17].*

In a zero-sum game, every action which changes utility affects both players in the game. For every loss in one player's utility, the other player gains an equal amount of utility. Zero-sum games are a subset of *constant-sum* games. In contrast with zero-sum, a constant-sum game will have an action a where $u_1(a) + u_2(a)$ is equal to a constant c [17].

Example 2.1.2. The card game War is a zero-sum game. In War, two players play the top card of their deck simultaneously. Whoever plays the higher value card takes both cards and places them at the bottom of their deck. In the event of a tie,

both players play two additional cards face-down and a third card face-up on top of the original tie. The winner of this exchange takes all cards in play. The utility of each card can be measured by its face value, and the utility of each player is equal to the sum of the utility of all cards in their deck. Suppose that there is a round where Player 1 plays a 10 and Player 2 plays a 7. Player 1 takes the 7, gaining 7 utility points while Player 2 loses those same 7 points. Let $u_1(a)$ represent the utility for player 1 of an interaction a and $u_2(a)$ represent the utility for player 2 of an interaction a. Thus, if player 1 plays an 10 and player 2 plays a 7, player 1 will keep the 10 and take the 7. The utility $u_1((10,7)) = 7$ since player 1 gained a 7 and $u_2((10,7)) = -7$ since player 2 lost a 7. Thus, $u_1(a) + u_2(a) = 7 - 7 = 0$. This holds after every play in the game, so War is a zero-sum game.

In comparison, a non-zero-sum game does not have an equal exchange of utility. Interactions between players are unequal. One player may gain five utility points while another player loses three utility points, or one player may gain utility with no effect on other players.

Example 2.1.3. Golf is not a zero-sum game. Suppose two players are competing on the same course for the best score. In golf, players are scored by the number of strokes it takes to reach each hole, with lower scores being more valuable than higher scores. If Player 1 takes two strokes on a particular hole, this only affects Player 1's score/utility; Player 2's utility is unaffected.

When games can be expressed in what is known as the *normal form*, this zero-sum quality affects the strategies and solutions of these games.

2.2 Normal Form Games

In the vast majority of human choices, a person will attempt to maximize their utility function; they will perform actions that benefit themselves. When there are two or more agents trying to maximize their utility, the interactions and choices between these agents, or players, become more and more complex. These interactions can be studied with game theory using the *normal* or *strategic* form.

Definition 4. *A finite normal-form game is a tuple* (N, A, u) *where:*

- N *is a finite set of n players, indexed by* i
- $A = A_1 \times \cdots \times A_n$*, where* A_i *is a finite set of actions available to player i. Each vector* $a = (a_1, \cdots, a_n) \in A$ *is called an action profile.*
- $u = (u_1, \cdots, u_n)$ *where* $u_i : A \mapsto \mathbb{R}$ *is a real-valued utility function for player i. [17]*

In an arbitrary normal-form game, there are n different players contained in the set N, where n is a positive integer. For each of these players, there is also a finite set of actions that they can take. Player N_i will have the action set A_i. The complete action set A is determined by the Cartesian product of the action sets of each player. Within A, a vector a is the action profile; an action profile, or outcome, represents the choice or choices that each player makes [9]. The collection of mappings u gives each action profile a particular value; these values determine which outcome is preferred by a player. In normal-form games, players are utility-maximizing agents; they will gravitate towards action profiles that give them higher utility. An example of a normal-form game can be seen in the quintessential "Prisoner's Dilemma" example.

Example 2.2.1. Two criminal partners are arrested on two different charges. The criminals are interviewed separately by the police. While in custody, each prisoner has a choice: they can either cooperate with their accomplice, or defect and testify

to the police. Without a confession, the police only have enough evidence to convict them on the lesser of the two crimes, which has a sentence of 1 year in prison. If the police can get a confession, they will be able to convict for the greater crime, which has a 3 year sentence. If one prisoner defects, the police are prepared to drop that prisoner's charges, and they will get no prison time. Their accomplice, on the other hand, will get the full 4 year total. If both prisoners defect they must both serve time; however, the judge will reward their cooperation with only a 3 year sentence. If both prisoners cooperate with each other (i.e. neither one confesses to the police), then they will both be convicted for the lesser crime and serve the 1 year sentence. The decisions in this game are expressed in a 2x2 matrix, with one player's choices represented in the columns and the other player's choices represented in the rows of said matrix.

		Player 2	
		C	D
Player 1	C	$-1, -1$	$-4, 0$
	D	$0, -4$	$-3, -3$

Figure 2.1: The Prisoner's Dilemma, represented in a normal form matrix [17].

In Figure 2.1, the column labeled C represents prisoner 2 cooperating with their partner, while the column labeled D represents prisoner 2 defecting, telling the police that their partner is guilty. The rows C and D represent the same choices for prisoner 1. For this game, A_i would be the set $\{C, D\}$, as those are the two choices that each prisoner can make. Thus, $A = A_1 \times A_2 = \{(C, C), (C, D), (D, C), (D, D)\}$, or all four possible outcomes for this game. An action profile a in this game would then be any of the ordered pairs in A; (C, D), for instance, is an action profile. The ordered pairs

within the matrix represent the values of each player's respective utility function in that outcome. In this case, their utility is the number of years in jail to which a player is sentenced. Since the players do not want to spend any time in prison, these utility values are negative; outcomes with less prison time have higher utility. The mapping functions in u correspond to the leniency of the courts; unfortunately, the quantitative functions in u that the courts use in this example are unknown, and only the values at these specific points are known. When both prisoners cooperate, they both get one year in prison; their sentence is lighter without their confession. If only one prisoner confesses, that player gets off with no time in prison, while the other player gets four years in jail. If both prisoners rat out their partner for committing the crime, both prisoners get three years in prison.

The prisoner's dilemma is not a zero-sum game. Supposing prisoner 1 chooses to defect (D), prisoner 2 could increase their utility from -4 to -3 by also defecting - a $+1$ change in utility for prisoner 2, but a -3 change in utility for prisoner 1.

From the perspective of an individual player, they must consider their options before proceeding. In considering their options, the best strategy can be determined using a game theory solution. In this example, the best strategy of a player would be trivial if that player also knew which strategies the other player(s) were adopting [17]. This would be a *pure strategy* option, where the player selects a single action and always that action. The other option is for a player to create a *mixed strategy* profile by introducing randomness into their own choice.

Definition 5. *Let* (N, A, u) *be a normal-form game, and for any set* X *let* $\Pi(X)$ *be the set of all probability distributions over* X*. The set of mixed strategies for player* i *is* $S_i = \Pi(A_i)$*, and the probability an action* $a_x \in A_i$ *will be played is denoted* $s_i(a_x)$*. A mixed-strategy profile is the Cartesian product of individual strategy sets* $S_1 \times \cdots \times S_n$ *[17].*

For an example of both pure and mixed strategies, consider a game where one player offers the other a choice of two different toys.

Example 2.2.2. There are two players and two presents containing toys. One player is designated the offering player, and the other player is designated the choosing player. The choosing player wants one of the toys, but not the other. The offering player picks which box to place each toy in, and the choosing player selects which present they want to open.

Suppose that the toys are placed in transparent boxes, and thus the contents of the boxes are not hidden. The offering player places each toy in a box, and the choosing player picks whichever toy gives them the most utility. In this case, the decision made by the offering player is virtually meaningless. It does not matter whether the toy desired by the choosing player is in box a or box b, since they can see which transparent box contains the toy they want. In this case, both the offering player and the choosing player use a pure strategy. Suppose instead that the toys are inside gift boxes - one red, one green. Once again, the offering player uses pure strategy to decide in which box to place each toy; they make a choice at the beginning of the game and stick with that choice. But, since the choosing player does not know which box contains their desired toy, they have a 50-50 chance of picking the toy they want. Thus, any pure strategy they might formulate (such as always picking the red box) will not be guaranteed to maximize the choosing player's utility. Thus, since it is unclear which decision is best, a mixed strategy is more appropriate. A pure strategy is a special case of mixed strategy, where the probability $s_i(a_x) = 1$ for a particular value of x and $s_i(a_k) = 0$ for all values of $k \neq x$. With the mixed strategies determined, the players can find their *best response* to a mixed-strategy profile.

Definition 6. *A best response for player i is a mixed strategy s_i^*. Let S be the mixed-strategy profile for a normal-form game, and s_{-i} be the set of strategy profiles disjoint from player i's strategy profile s_i. The best response $s_i^* \in S_i$, such that $u_i(s_i^*, s_{-i}) \geq u_i(s_i, s_{-i})$ for all strategies $s_i \in S_i$ [17].*

In two-player games, like the toy choosing game, finding s_{-i} is simple: if $i = 1$, then $s_{-i} = s_2$, the strategy profile for player 2. In a three-player game with $i = 1$, s_{-i} would be $\{s_2, s_3\}$, the set containing player 2's strategy and player 3's strategy. For player i to determine their best response, they must find a mixed strategy which maximizes utility against s_{-i}. In rare cases, this mixed strategy will actually be a unique pure strategy. Most of the time, there will be an infinite number of best responses with two or more actions in s_i [17]. These actions must have the same value to the player, or else that player would want to minimize the probability of one such action. And therefore, since neither option is preferred over another, every possible combination of probabilities for actions in s_i is a best response.

For the toy choosing game, the mixed-strategy profile is the Cartesian product of Player 1 and Player 2's strategy sets. To better illustrate the mixed-strategy profile, assume Player 1, the player offering the gifts, chooses to put the desired toy in box a or b by flipping a coin, rather than by pure strategy. Player 1's mixed-strategy profile S_1 is therefore $\{.5a_1, .5b_1\}$. Player 2, the selecting player, chooses either box a or box b. Thus, Player 2's mixed-strategy profile S_2 is $\{.5a_2, .5b_2\}$. The mixed-strategy profile for the entire game is therefore $S = \{(.5a_1, .5a_2), (.5a_1, .5b_2), (.5b_1, .5a_2), (.5b_1, .5b_2)\}$. In this example game, any element in S is a best response. Player 1 has no preference for either box a or box b; Player 2 does not know which box they prefer, and treats both equally.

Without knowing the strategy of other players, the most an agent can do is choose a strategy that is better than other strategies in all possible scenarios. This is known as the Nash equilibrium.

Definition 7. *A strategy profile $s = (s_1, \ldots, s_n)$ is a Nash equilibrium if, for all agents i, s_i is a best response to s_{-i} [17].*

To find the Nash equilibrium for a player in the prisoner's dilemma example, the action of the other player is assumed to be a constant, pure strategy. So, while

determining Player 1's Nash equilibrium, we first assume that Player 2 always cooperates with Player 1. This assumption limits Player 1's decision to a single column in the normal form.

		Player 2
		C
Player 1	C	$-1, -1$
	D	$0, -4$

Figure 2.2: Player 1's best response in the Prisoner's Dilemma, assuming Player 2 cooperates with Player 1.

In Figure 2.2, it is trivial to find the utility-maximizing strategy. In this case, the best response to Player 2's cooperation is for Player 1 to defect and accuse their partner: one year in prison by cooperating or no time in prison by defecting.

		Player 2
		D
Player 1	C	$-4, 0$
	D	$-3, -3$

Figure 2.3: Player 1's best response in the Prisoner's Dilemma, assuming Player 2 will defect against Player 1.

Next, we assume that Player 2 always defects. In Figure 2.3, it is again trivial to find the utility-maximizing strategy for Player 1. Assuming Player 2 will defect to the police, Player 1's best response is to defect as well: 3 years in prison by defecting

or 4 years in prison by cooperating. So, for all of Player 2's possible strategies, Player 1's best response is to defect.

Using the same process for Player 2, we first assume that Player 1 uses pure strategy, and that their pure strategy is to cooperate with Player 2.

		Player 2	
		C	D
Player 1	C	$-1, -1$	$-4, 0$

Figure 2.4: Player 2's best response in the Prisoner's Dilemma, assuming Player 1 will cooperate with Player 2.

In Figure 2.4, it is trivial to find the utility-maximizing strategy. In this case, the best response to Player 1's cooperation is for Player 2 to defect and accuse their partner: one year in prison by cooperating or no time in prison by defecting.

		Player 2	
		C	D
Player 1	D	$0, -4$	$-3, -3$

Figure 2.5: Player 1's best response in the Prisoner's Dilemma, assuming Player 2 will defect against Player 1.

Next, we assume that Player 1 always defects. In Figure 2.5, it is again trivial to find the utility-maximizing strategy for Player 2. Assuming Player 1 will defect to the police, Player 1's best response is to defect as well: 3 years in prison by defecting or 4 years in prison by cooperating.

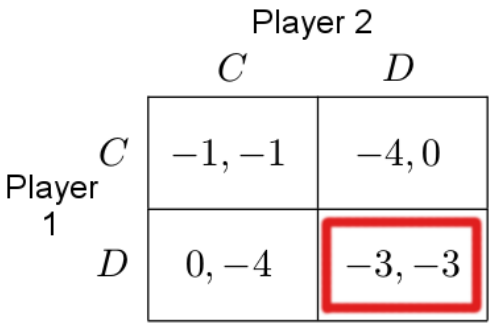

Figure 2.6: The Nash Equilibrium of the Prisoner's Dilemma: Both prisoners defect and both are sentenced to three years in prison.

So, for all of Player 1's possible strategies, Player 2's best response is to defect. Thus, the Nash equilibrium is that both prisoners will defect and get 3 years in prison, as seen in Figure 2.6. A more robust proof of the Nash equilibrium for the Prisoner's Dilemma is shown below.

Theorem 2.1.

Assume a normal form game $G = (N, A, u)$, $N = \{n_1, n_2\}$, *and* $A = \{A_1 \times A_2\} = \{c_1, d_1\} \times \{c_2, d_2\}$. *If the set of utility functions* $u = \{u_1 = (c_1, c_2) \mapsto -1, (c_1, d_2) \mapsto -4, (d_1, c_2) \mapsto 0, (d_1, d_2) \mapsto -3\}, u_2 = (c_1, c_2) \mapsto -1, (c_1, d_2) \mapsto 0, (d_1, c_2) \mapsto -4, (d_1, d_2) \mapsto -3$, *then the Nash equilibrium of G is the strategy profile* (d_1, d_2), *where* d_1 *is the choice for player 1 to defect and* d_2 *is the choice for player 2 to defect.*

Proof. Let $G = (N, A, u)$ be a normal-form game. $N = \{n_1, n_2\}$ is the set of players in the game. $A = A_1 \times A_2$ is the set of actions in the game, where $A_1 = \{c_1, d_1\}$ is the set of actions available to player 1, and $A_2 = \{c_2, d_2\}$ is the set of actions available to player 2. Therefore, $A = \{(c_1, c_2), (c_1, d_2), (d_1, c_2), (d_1, d_2)\}$. The utility function $u = (u_1, u_2)$. $u_1 = (c_1, c_2) \mapsto -1, (c_1, d_2) \mapsto -4, (d_1, c_2) \mapsto 0, (d_1, d_2) \mapsto -3$, and $u_2 = (c_1, c_2) \mapsto -1, (c_1, d_2) \mapsto 0, (d_1, c_2) \mapsto -4, (d_1, d_2) \mapsto -3$.

Let s_1^* be a mixed strategy for n_1 where n_1 chooses action d_1 with a probability of 1. Since there are only two players, the set of strategy profiles disjoint from s_1 is the

same as s_2, the set of strategy profiles for n_2. Thus, n_1 has s_1^* as a best response to s_2 if $u_1(s_1^*, s_2) \geq u_1(s_i, s_2)$ for all possible actions $s_i \in A_1$ and all strategies in s_2.

$u_1(d_1, c_2) \geq u_1(c_1, c_2) \Rightarrow 0 \geq -1$

$u_1(d_1, c_2) \geq u_1(d_1, c_2) \Rightarrow 0 \geq 0$

$u_1(d_1, d_2) \geq u_1(c_1, d_2) \Rightarrow -3 \geq -4$

$u_1(d_1, d_2) \geq u_1(d_1, d_2) \Rightarrow -3 \geq -3$

s_1^* is greater than or equal to all possible actions in A_1 and for all strategies in s_2. Thus, $s_1^* = d_1$ is the best response for player n_1.

Let s_2^* be a mixed strategy for n_2 where n_2 chooses action d_2 with a probability of 1. Since there are only two players, the set of strategy profiles disjoint from s_2 is the same as s_1, the set of strategy profiles for n_1. Thus, n_2 has s_2^* as a best response to s_1 if $u_2(s_2^*, s_1) \geq u_2(s_i, s_1)$ for all possible actions $s_i \in A_2$ and all strategies in s_1.

$u_1(c_1, d_2) \geq u_1(c_1, c_2) \Rightarrow 0 \geq -1$

$u_1(c_1, d_2) \geq u_1(c_1, d_2) \Rightarrow 0 \geq 0$

$u_1(d_1, d_2) \geq u_1(d_1, c_2) \Rightarrow -3 \geq -4$

$u_1(d_1, d_2) \geq u_1(d_1, d_2) \Rightarrow -3 \geq -3$

s_2^* is greater than or equal to all possible actions in A_2 and for all strategies in s_1. Thus, $s_2^* = d_2$ is the best response for player n_1.

A strategy set $(s_1, \ldots, s_n)$ is a Nash equilibrium if s_i is the best response for all players i. Therefore, the strategy profile (d_1, d_2) is a Nash equilibrium for the game G, the Prisoner's Dilemma. □

Not all games have a Nash equilibrium; take the normal-form representation of rock-paper-scissors, for instance.

Example 2.2.3. In Rock-Paper-Scissors, two players choose a hand sign at the same time. If both play the same sign, the round is a draw. If two different signs are played, a winner is determined using the following rules: rock beats scissors, scissors beat paper, and paper beats rock. Let the Players 1 and 2 have utility functions which

award 1 point for winning the game, -1 point for losing the game, and 0 points for a draw. Notice that these values create a zero-sum game; a win for Player 1 comes with a loss for Player 2. We position Player 1 on the left side of the matrix, and Player 2 on the top of the normal-form matrix.

Rock-Paper-Scissors		Player 2		
		Rock	Paper	Scissors
Player 1	Rock	(0, 0)	(-1, 1)	(1, -1)
	Paper	(1, -1)	(0, 0)	(-1, 1)
	Scissors	(-1, 1)	(1, -1)	(0, 0)

Figure 2.7: The normal-form representation of the game Rock-Paper-Scissors

Every row and every column in Figure 2.7 has the same three results, and none of the choices is a winning strategy for more than one situation. Rock may beat Scissors in some games, but it loses to Paper in others. If the Nash equilibrium of Player 1 is investigated, each possible action by Player 2 results in a different best response for Player 1. For every situation where Player 1's choice provides the most utility, there exists another, equally-likely situation where the same choice would provide the least utility (in this case, negative utility) and a third equally likely situation where that choice provides no utility. Therefore, there can be no Nash equilibrium.

2.2.1 Non-Simultaneous Games

It is important to recognize in these examples that both players are making their choice simultaneously. Games without simultaneous actions are not so easily expressed in normal form. For example, take a game of tic-tac-toe. For simplicity, assume that the game is played with a 2x2 board instead of a 3x3 board. Assume

the utility function awards players 1 point for winning the game, -1 point for losing the game, and 0 points for a draw. Let UL be a piece in the upper left square, UR the upper right square, LL the lower left square, and LR the lower right square. We position Player 1 on the left side of the matrix, and Player 2 on the top of the matrix. Assume Player 1 has the first turn.

2x2 tic-tac-toe		Player 2 UL	UR	LL	LR
	UL	x	(0, 0)	(0, 0)	(0, 0)
Player 1	UR	(0, 0)	x	(0, 0)	(0, 0)
	LL	(0, 0)	(0, 0)	x	(0, 0)
	LR	(0, 0)	(0, 0)	(0, 0)	x

Figure 2.8: The first two moves of a tic-tac-toe variant using a 2x2 game board.

Figure 2.8 shows the normal form for the first two moves of this game. Since each player has only placed a single piece, neither one has won yet; thus, the utility for each space is (0,0). There are four spaces in the matrix marked with an 'x'. These spaces are infeasible; it is against the rules for someone to play on the same square as someone else. Both players have made their first move, and it is Player 1's turn once again.

2x2 tic-tac-toe		Player 2 UL	UR	LL	LR
Player 1	(UL, UR)	x	x	(1, -1)	(1, -1)
	(UL, LL)	x	(1, -1)	x	(1, -1)
	(UL, LR)	x	(1, -1)	(1, -1)	x
	(UR, UL)	x	x	(1, -1)	(1, -1)
	(UR, LL)	(1, -1)	x	x	(1, -1)
	(UR, LR)	(1, -1)	x	(1, -1)	x
	(LL, UL)	x	(1, -1)	x	(1, -1)
	(LL, UR)	(1, -1)	x	x	(1, -1)
	(LL, LR)	(1, -1)	(1, -1)	x	x
	(LR, UL)	x	(1, -1)	(1, -1)	x
	(LR, UR)	(1, -1)	x	(1, -1)	x
	(LR, LL)	(1, -1)	(1, -1)	x	x

Figure 2.9: The full normal form representation of a 2x2 tic-tac-toe variant

By nature of this compressed game board, the first player to place their second piece (in this case, Player 1) cannot lose; all legal moves lead to victory. Even in this simplified game, the normal form in Figure 2.9 quickly becomes convoluted and cluttered. While the infeasible actions are fairly self-evident in Figure 2.8, illegal moves are less obvious in Figure 2.9.

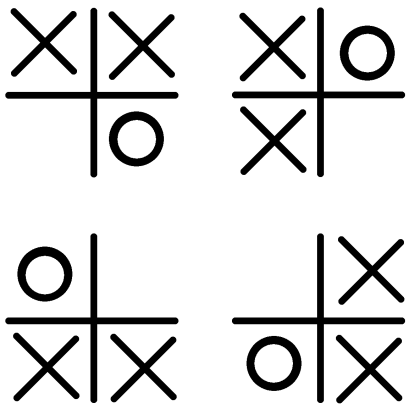

Figure 2.10: Rotational symmetry in a 2x2 tic-tac-toe game

There are also repeated game configurations: the outcome {(UL, UR), (LR)} is rotationally symmetrical to {(LL, UL), (UR)}, {(LR, LL), (UL)}, and {(UR, LR), (LL)}, as is evident in Figure 2.10. Furthermore, the game {(UL, UR), (LR)} is identical to {(UR, UL), (LR)}. With so many extraneous spaces in the normal-form representation, it is no longer sufficient to represent player choices for games with sequential actions. Instead, these games are better represented in extensive-form trees.

2.3 Extensive Form Trees

In a turn-based, or an extensive-form game, players do not make simultaneous actions. This leads to larger variety in gameplay. A simultaneous game like Rock-Paper-Scissors relies on random chance for which hand sign a player happens to pick; tic-tac-toe relies on players anticipating the future choices their opponents could make. There are two types of extensive-form games: perfect-information and imperfect-information. In perfect-information games, all players know which turn of the game they are at, while imperfect-information games have turns that are indistinguishable from others based on information available to the player.

Example 2.3.1. The card game Go Fish is an imperfect-information game. In Go

Fish, players are dealt a hand of playing cards. On their turn, one player asks another player for a specific card in their hand; for example, Player 1 may ask if Player 3 has a "5" in their hand. If so, Player 3 must hand over their "5" to Player 1. However, Player 1 can only ask about a card if Player 1 has one of the other "5" cards in their hand. If Player 3 does not have a "5," then Player 1 must draw another card from the draw pile. Once a player has all four of a particular card value, the cards form a book and are removed from play. At the end of the game, the player with the most books wins.

Go Fish is an imperfect information game since any individual guess is mostly indistinguishable from another. Even if Player 1 determines that Player 3 has no "5's," that fact could change when Player 3 draws another card. We will focus on perfect-information games.

These games are represented in graph-theoretic trees. Specifically, extensive-form games are represented with *rooted trees*.

Definition 8. *A rooted tree is a pair* (V, E)*, where* V *(known as the vertex set) is a finite set and* E *(known as the edge set) is a binary relation on* V *[11]. Elements of* V *are referred to as vertices or nodes, and elements of* E *are referred to as edges. Trees are acyclic, meaning that no cycles exist in the graph; there is no path which moves away from a node* a *by edge* e *and returns to* a *by a different edge. Trees are connected, meaning every vertex in* V *can be reached from all other vertices in* V*. One element* $v \in V$ *is denoted as the root.*

These types of graphs are called trees due to the visualization in a tree-like diagram, showing the *root* and *children*. In these trees, such as the one in Figure 2.11, the node at the top of the tree is designated the root, and all nodes connected by a single edge to the root are the root's children. A *subtree* can be created by taking an arbitrary node in the tree and treating that as a root node, with all nodes above the new root excluded from the subtree.

Definition 9. *A finite perfect information game in extensive form is a tuple* $G = (N, A, H, Z, \chi, \rho, \sigma, u)$ *where:*

- N *is a set of n players;*
- A *is a single set of actions;*
- H *is a set of non-terminal, choice nodes;*
- Z *is a set of terminal nodes, disjoint from* H*;*
- $\chi : H \mapsto 2^A$ *is the action function, mapping a set of possible actions to each choice node;*
- $\rho : H \mapsto N$ *is the player function, mapping to each non-terminal node a player who makes an action at said node;*
- $\sigma : H \times A \mapsto H \bigcup Z$ *is the successor function, mapping a choice node and an action to a new choice or terminal node.* $\forall h_1, h_2 \in H$ *and* $a_1, a_2 \in A$*, if* $\sigma(h_1, a_1) = \sigma(h_2, a_2)$*, then* $h_1 = h_2$ *and* $a_1 = a_2$*;*
- $u = (u_1, ..., u_n)$*, where* $u_i : Z \mapsto \mathbb{R}$ *is a real-valued utility function for each player i on a terminal node Z [17].*

As in the normal-form definition, there are n players in the game, contained in the set N. A is the set of all actions that could be performed in the course of the game. The set H contains any and all choices or turns that do not end the game, while Z contains all choices or turns that do end the game. The mapping function χ allocates the choices in H a subset of possible actions from the set A, and the mapping function ρ assigns these choices to the players who make them. σ takes the actions assigned with χ and uses them to connect various nodes in H and Z together, creating the sequence of events in the game. Finally, u contains utility functions for

each player that determine their individual utility at each of the terminal nodes in Z. An example of the extensive-form tree can be seen in The Sharing Game.

Example 2.3.2. In The Sharing Game, a variation of the toy-offering game in Example 2.2.2, two children receive two identical presents from their parents, both equally valuable to the children. Player 1 suggests one of three splits: Player 1 receives both presents, Player 2 receives both presents, or each player receives one present. Once the split has been suggested, Player 2 chooses to accept the split or not. If the split is declined, neither player receives any presents.

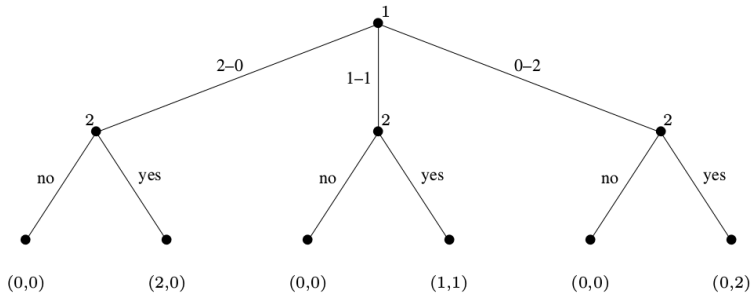

Figure 2.11: An extensive-form tree for The Sharing Game [17]

Figure 2.11 shows six leaf nodes of this tree - the elements of the set Z - while the other nodes are contained in H. The H nodes are labeled either 1 or 2, denoting from the ρ function which player is making the choice at that node. The Z nodes each have a pair, denoting the utility each player will gain if the game ends in that particular way. The actions A are denoted in the lines connecting the nodes together. In this game, the three actions from the root node are the methods of sharing two presents between two people. The second player, once the first player has distributed the presents in a particular way, chooses whether to accept the distribution or not.

To explore this definition, return to the game of 2x2 tic-tac-toe. Let $N = \{N_1, N_2\}$ be the set of players in the game. In this game, A is the set of open spaces on which

a player can place their symbol; for the first move of the game, A is the set of 4 open spaces on the board, for the second move, A is the set of the remaining 3 spaces. For a game of tic-tac-toe, H contains configurations of the board where neither player has two in a row and there are still open squares on the board. Z, on the other hand, contains the board configurations where one player has two in a row or all spaces have been claimed. For tic-tac-toe, the χ function removes illegal actions from a node, which for tic-tac-toe would be playing on an occupied square. Thus, the χ function leaves the choice nodes at each successive turn with fewer and fewer available actions. Since there are only two players, the ρ function alternates between players after each action.

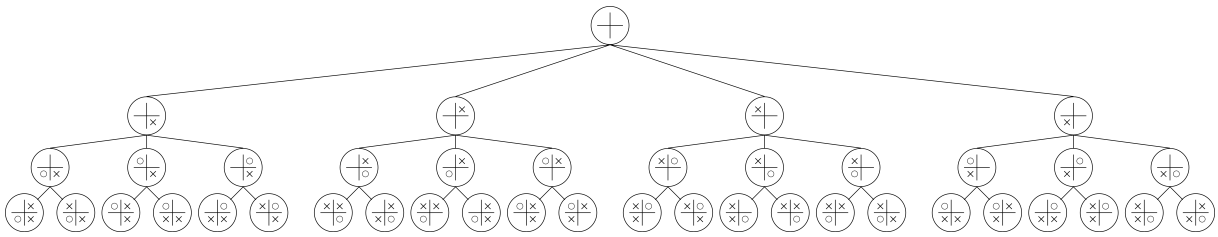

Figure 2.12: The extensive-form representation of a 2x2 tic-tac-toe game

Figure 2.12 shows the extensive-form version of the 2x2 tic-tac-toe variant in Figure 2.9. In comparison to the normal-form representation, the extensive-form tree can convey the sequence and flow of the game more clearly. Since the χ function of an extensive-form game only maps possible actions, the tree does not have the same infeasible nodes as the normal-form representation. This tree has the same issues of rotational symmetry as in the normal form, but in this particular example, the four subtrees connected to the root are all rotationally symmetrical to each other.

2.3.1 Equilibria in Extensive-Form

Nash equilibria can be found in extensive-form games as well, and are defined the same as in normal-form games. However, a pure strategy in an extensive-form game requires the player's strategy to contain the decision at every choice node, even if that node is unreachable by other choices in said strategy.

Definition 10. *Let $G = (N, A, H, Z, \chi, \rho, \sigma, u)$ be a perfect-information extensive-form game. The pure strategies of player i consist of the Cartesian product $\Pi_{h \in H, \rho(h)=i} \chi(h)$ [17].*

Given a perfect-information game, a player's pure strategy is the set of choices mapped by χ from each of that player's choice nodes h. It is a complete specification of the choices that a player determines they should make [17]. Since the actions of all other players are known in an extensive-form game, it is not necessary to introduce probabilities to determine a Nash equilibrium. Instead, Nash equilibria can be found using backwards induction.

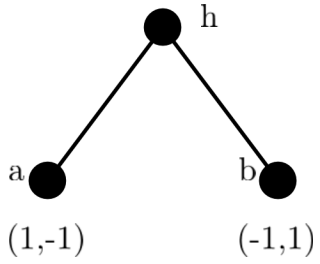

Figure 2.13: A small extensive-form tree. The outcome is dependent on the player mapped by $\rho(h)$.

Backwards induction determines a dominant strategy for a particular game. This strategy works on the assumption that players act optimally, and will thus always choose a leaf node which maximizes their utility. Therefore, a decision node n with a set of leaf nodes as its children is equivalent to the leaf which maximizes the utility of the deciding player. For example, examine the extensive-form tree for a two-player game in Figure 2.13. with a root and two child nodes. Both of these children are leaf or terminal nodes. At the leaf node a, Player 1 has positive utility

while Player 2 has negative utility; at leaf node b, Player 2 has positive utility and Player 1 has negative utility. Since both players are utility-maximizing agents, the choice of the outcome is determined by whichever player happens to be making the decision at the root, as mapped by $\rho(h)$. Suppose that Figure 2.13 is a subtree in a larger game. Further suppose that, in this larger game, the subtree in Figure 2.13 always has Player 1 making the choice at node h. Given this scenario, Player 1 will always perform the action that gets them to node a. Thus, the tree in Figure 2.13 is equivalent to the leaf node a, with b being removed from the tree entirely. If the entire tree for a game is mapped out, then this process can be repeated all the way up to a root node; at that point, the outcome of the game can be determined before the game even begins. The pseudocode below describes the process of finding a subgame-perfect equilibrium from the bottom of an extensive tree upward.

```
1   Procedure BackwardInduction(node h)
2   if(h in Z)
3       return u(h)
4       //If h is a leaf node, return the utility of that
            outcome
5   end if
6   for(a in χ(h))
7       util_at_child = BackwardInduction(σ(h,a))
8       //Recursively check each child of h
9       if(util_at_child_ρ(h) > best_util_ρ(h))
10          best_util = util_at_child
11          //best_util holds the node which maximizes the
                utility of player ρ(h)
12      end if
13  end for
```

```
14    return best_util
15    end Procedure
```

Listing 2.1: The Backward Induction algorithm for extensive-form game trees [17].

The algorithm in Listing 2.1 has a base case at line 2, where h is in the set of terminal nodes in Z, or the leaves of the extensive-form tree. At these leaves the game is over, and the utility of that outcome is returned by the algorithm. When the node in question is not a terminal node, the algorithm checks the utility of each of that node's children at line 6. Since the child nodes may have their own children, line 7 recursively checks each child and gets the utility of that node. In lines 9 and 10, the backwards induction algorithm determines which node provides the player at node h (as determined by the mapping function ρ) the maximum amount of utility. This node is returned in line 14.

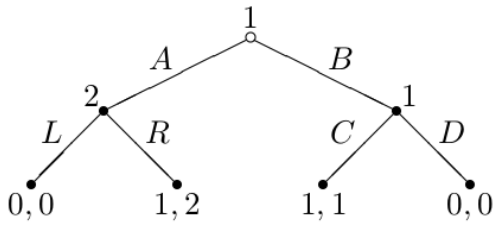

Figure 2.14: An extensive-form game tree which cannot be completely solved with backward induction.

Not all game trees can be reduced with the backwards induction algorithm. In Figure 2.14, two players play such a game.

Example 2.3.3. Player 1 has a choice between A and B on their first turn. If Player 1 chooses B, then they make a second choice between C and D. Player 2 only has one choice: if Player 1 chooses A, then Player 2 chooses between L and R.

Using the backwards induction algorithm on the left subtree of Figure 2.14, we find that the outcome (1, 2) - the leaf node reached by the choices $\{A, R\}$ - is preferred

by Player 2 to the outcome (0, 0) - the leaf node reached by the choices $\{A, L\}$. Thus, the left child of the root - Player 2's choice node - is replaced with the leaf node with utility (1, 2) by the backwards induction algorithm. On the right subtree in Figure 2.14, Player 1 makes a similar choice between (1, 1) - from the choices $\{B, C\}$ - and (0, 0) - from the choices $\{B, D\}$. Thus, the right child of the root is replaced with the leaf node with a utility value of (1, 1). Now, the algorithm can find the expected utility value of the root node. Since Player 1 is making the decision at the root, their choices have equal utility values: 1 utility from (1, 1) and 1 utility from (1, 2). Thus, Player 1 has no preference to choosing action A or B. However, Player 2 does have a preference in this matter, but no way to influence Player 1's choice. Backwards induction is insufficient in this game to predict, with complete accuracy, whether the final outcome of the game will be (1,2) or (1,1).

2.4 Other Games

While normal-form and extensive-form games are the two largest categories of games, there are other types of games independent of the former two. One such type is *repeated games* and *stochastic games*.

Compared to the games previously discussed, referred to as *one-shot games*, repeated games involve the same players playing the same game multiple times [5]. Repeated games deal with the notion of trust, and what players expect their fellow players to do. After multiple iterations of the same game, players may start to predict their opponent's choices. However, these repetitions also encourage certain behaviors. For example, take the card game poker.

Example 2.4.1. In poker, players are dealt a hand of playing cards and make bets on their hands; for instance, a player may bet $100 that their hand has a higher rank than the hands of their opponents. These bets go into a communal pot, which is

awarded at the end of the game. After a bet, the other players then have a chance to call the bet (adding in $100 of their own to the bet), raising the bet (for instance, betting $150 that their hand has the highest rank), or folding (exiting the game with no chance to regain the money in the pot). After a certain number of betting rounds, all players who have not folded show their hands, and the player with the highest ranking hand wins the money in the pot. The number of cards dealt, whether cards are dealt face-up or face-down, and the rank of various hands depend on the poker variant.

Suppose in the poker example that the same group of players play ten rounds against each other. After the first few rounds, players can infer some of the strategies employed by other players by earlier repetitions of the game. For instance, suppose a player always bluffs, pretending that they have a high-ranking hand even when they have a worthless hand. When this player, who we denote as player A, first bluffs in this manner, they may convince other players to fold. Thus, player A may win some early rounds by deception. However, if in repeated games other players recognize player A's strategy, they will become less susceptible to player A's lies. In this way, the repeated nature of the game creates incentives fundamentally different from those of a one-shot game [5].

There is a repeated version of the Prisoner's Dilemma as well. As established in Figure 2.6, the Nash equilibrium of the game is for both players to defect and to both serve three years in jail. This equilibrium comes about because players value the short-term goal of reaching a (0, -4) utility outcome [9] and both players are assumed to play optimally. However, it is possible that, without loss of generality, player 2 will not play optimally and choose to cooperate instead of defect as the Nash equilibrium would suggest. Since the original version of the Prisoner's Dilemma is a one-shot game, this scenario would greatly benefit player 1. However, in a repeated game, the defection strategy is no longer optimal.

In stochastic games, the actions taken by players has an effect on the *environment* of the game [19]. At each step of the game, play moves from position to position. These positions are determined by *transition probabilities*, which both players in the game control [15]. At any position of the game, each combination of choices for the two players has a probability greater than zero of ending the game and a probability of moving to another position of the game. For example, take the party game Twister.

Example 2.4.2. In Twister, players contort themselves on a large playing mat. The mat has four columns of dots, with each column's dots in a different color. A neutral entity uses a spinner, which is split into quarters: right hand, left hand, right foot, and left foot. Each of these sections is further split into the four colors. When this neutral entity spins the spinner, players must perform the indicated action; for example, if the spinner lands on "right foot red," all players must move their right foot onto a red dot. Players continue until they fall down, at which point they are out of the game.

At the starting position of the game, no players have any of their limbs on the playing mat. Suppose without loss of generality that the first turn is for players to move their right foot to a red dot. The probability of the game ending at this position will depend on which red dots players use. If players move their feet to distant red dots, so that their legs are spread far apart, it is more likely that the game will end than if players choose closer red dots. However, there is still the possibility that a player will slip and fall on the first step, even at the closest red dots.

Assume that play has continued further into the game, to an arbitrary position of limbs. The environment in Twister is dependent on the contortions into which players have put themselves. Thus, players have a preference to move themselves in ways which create more stable positions. Since there are a limited number of dots to which players can move, these positions get in the way of other players,

and can cause them to lose their balance. Thus, players must find a metaphorical balance between keeping themselves stable and interfering with their opponents to be successful at the game.

2.5 Conclusion

Game theory provides a number of techniques and methods to find optimal strategies for games. These techniques can be applied to a wide variety of game types, including both games where players act simultaneously and games where players take turns acting. Game theory can also explain games where choices in one game affect choices in later iterations of the game, and games where a player's action affects the environment of the game. For games where players take turns, an extensive tree is used to represent the decisions made by a player at each turn of the game. An extensive tree can be solved using the backwards induction algorithm, at which point the outcome of the game is known from the start.

CHAPTER 3

Machine Learning

3.1 Decision Trees

The goal of a machine learning problem is, of course, to learn; given a task with a particular performance measure, the performance can be improved through some kind of training or experience. Often, this involves evaluating a general case for the task from a set of specific examples [6]. One of the more common machine learning techniques is the decision tree. It can approximate discrete-valued functions [6] and automatically classify data [14]. Generally, decision trees focus on problems with attribute-value pairs; an attribute *Temperature* may have the possible values *Hot, Cold, Warm, Cool* [6].

When creating a decision tree, one must begin with example data to analyze. This may be historical data, looking at the same type of results from previous years in order to predict current trends, or a self-contained set of data to be used on similar inputs. In order to form an effective tree, these data are split into two groups: a training set, and a testing set. The training set is used to create the tree, while the testing set is used to find the approximate error of the tree. It is important that these two sets are disjoint; the error of a tree cannot be accurately determined if it isn't analyzing new data in the testing phase.

3.1.1 ID3 Algorithm

The ID3 algorithm for decision trees, like most decision tree algorithms, uses a top-down greedy search [6]. The algorithm begins by evaluating the input data by individual attributes in the training set. For each attribute, the algorithm attempts to classify the entire data set using only that attribute. Whichever attribute was the most successful on its own becomes the root of the tree. Then, for each possible value of that root attribute, a descendant node is created and the training set is split accordingly. Then, for each descendant node and its respective subset of training data, the process is repeated.

To find the best attribute at each node of the tree, the ID3 algorithm uses the statistical property information gain. Information gain utilizes entropy, or the relative impurity of a data collection.

Definition 11. *For a set of training data S with n different target values, Entropy(S)*$\equiv \sum_{i=1}^{n} -p_i log_2 p_i$*, where* p_i *is the proportion of S with a target value of i.*

Entropy calculations define 0log(0) as 0. The logarithm can have any base value, but two is often used; a base of 2 conveys the expected encoding length in bits [6]. Assuming the base value is 2, an entropy value of 0 occurs when all data set elements have the same target value i, and a set with a 50/50 split between two target values will have an entropy value of 1. Ideally, a decision tree classifies data into sets with low entropy, as this indicates an accurate tree structure.

In order to minimize entropy in the classified data, the information gain of each attribute is calculated. Information gain is defined as the expected reduction in entropy caused by splitting the training data by a particular attribute.

Definition 12. *For an attribute A and a set of training data S, the Information Gain(S, A)*$\equiv$ *Entropy(S)* $- \sum_{v \in Values(A)} \frac{\|S_v\|}{\|S\|}$ *Entropy*(S_v)*, where* $Values(A)$ *is the set of possible values for the attribute A, and* S_v *is the subset of S where A has value v.*

Each attribute A for a given node is tested, with each of its possible values used in the summation. For these values, the entropy of each subset S_v is calculated, then scaled by the size of S_v compared to S.

Month	Tickets	Weekend
3	25	0
8	45	1
8	25	0

Figure 3.1: A sample data set with three attributes and three entries

For example, consider the data set in Figure 3.1. The *Month* attribute is the numeric value of the calendar month, *Tickets* is the number of tickets sold to an event, and *Weekend* is a Boolean value whether the event is on a weekend or not. Suppose that the target attribute of this set is *Weekend*. The *Weekend* attribute has two values represented in the set: 0 and 1. Of the data points in Figure 3.1, only one took place on a weekend. Thus, the p_i for Weekend=True is $\frac{1}{3}$, which we round to .33. The p_i for Weekend=False is therefore $\frac{2}{3}$ rounded to .66. Recall that the entropy formula is Entropy(S)$\equiv \sum_{i=1}^{n} -p_i \log_2 p_i$. Let $i = 1$ be the attribute Weekend=True and $i = 2$ be the attribute Weekend=False. The entropy for the target attribute *Weekend* is therefore the sum $-.66\log_2(0.66) + -.33\log_2(.33)$, or approximately 0.9235. For this specific set, all attributes have two possible values, and will thus have a $\frac{1}{3}$ and $\frac{2}{3}$ split. In this example, the entropy will be the same for any target attribute.

To reduce this entropy, the ID3 algorithm finds the information gain for the two non-target attributes: *Month* and *Tickets*. Beginning with *Month*, the information gain is $0.9235 - \frac{1}{3}$(Entropy(v=3)) $+\frac{2}{3}$(Entropy(v=8)). At *Month* 3, the entropy of the set is 0, as it contains a single element. The entropy of the subset at *Month*=8 is 1, as the set has a 50/50 split along the target *Weekend* attribute. Thus, the information gain for *Month* is $0.9235 - \frac{2}{3}$, or 0.2568. Separating the data set along the *Month*

attribute does help in classifying the events by weekend, but the August events are now split 50/50.

The *Tickets* attribute has an information gain of $0.9235 - \frac{2}{3}$(Entropy(v=25)) $+\frac{1}{3}$(Entropy(v=45)). Since all elements in the subset of S where *Tickets*=25 have a target values of 0, the entropy is 0. The same is true of the subset where *Tickets*=45. Thus, the information gain from this attribute is 0.9235. This does not indicate that no information was gained, this indicates that all entropy was removed from the set. All subsets are homogeneous when separating subsets by *Tickets*.

Once the maximum information gain for a set has been determined, the ID3 algorithm uses the resulting subsets and recursively creates new subtrees for the subsets. This process continues until the entropy for the entire tree is 0. Once a tree has been created with the algorithm, it is simple to classify new data. Starting at the root of the tree, each target attribute is tested recursively, moving from one node to another down the tree [14].

```
1  Procedure ID3(TrainingSet S, TargetAttribute t, AttributeSet
       A)
2  create node Root
3  TestEntropy = True, TestAttributeValue = S[0]
4  for(i in S)  //Check whether the TrainingSet is uniform on t
5    if(TargetAttribute[i]!=TestAttributeValue) TestEntropy=
         False
6    //One of the elements has a different value for t
7    end if
8  end for
9  if(TestEntropy is True) return root with label value of t
10 //This is a single-node tree; all elements in S have the
     same t value
```

```
11 else if(A is EmptySet) return Root with mode(t)
12 //There are no more attributes to check, return most common
     t value
13 else
14 //Determine optimal split for S
15   OptimalInformationGain = 0
16   for(attr in A)
17     InformationGain = Entropy(S)
18     EntropySum = 0
19     for(val in attr)
20       S_v = {S:Value(attribute)=val} //S_v contains the
           elements of S which have val for attr
21       EntropySum = EntropySum + size(S_v)/size(S) * Entropy(
           S_v)  //Add entropy for each possible val for attr
22     end for
23     InformationGain = InformationGain - EntropySum
24     //This is the information gain from splitting S on attr
25     if(InformationGain > OptimalInformationGain)
26       OptimalInformationGain = InformationGain
27       //The maximum InformationGain value is stored
28     end if
29   end for
30   //The optimal information gain has been found on attr
31   Root.Label = attr
32   for(val in attr)
33     create Branch(val_i)
34     NewTrainingData = {s in S:Value(attr)=val_i}
```

```
35      if(NewTrainingData is EmptySet)
36        create Leaf with mode(t) on S
37      else
38        add ID3(NewTrainingData, t, A-attr) as subtree
39      end if
40    end for
41  end if
42  return Root
43  end Procedure
```

Listing 3.1: The ID3 Decision Tree algorithm [6].

In line 2 of the algorithm in Listing 3.1, the root of the tree (or subtree) is created. The first base case to test is whether or not the TargetAttribute t has the same value for every element in the TrainingSet S. This base case refers to a set S with an entropy of 0, that requires no further classification, and begins at line 5. The second base case, starting on line 15, occurs when no non-target attributes remain in A. Without attributes, there is no way to further classify S, and the most common value for t is used as the label for this node. This leaves the recursive case. The first step is to find the optimal information gain, and thus the next attribute used to split S. The loop at line 21 checks each attribute in A for the largest reduction in entropy, and the attribute that created this reduction is saved as the label for the Root node on line 38. Line 39's loop prepares the data to be analyzed by the next levels of the decision tree. First, a new branch is created linking Root and the to-be-created subtree, with each branch labeled with a possible value of the attribute used in the split. Training data is split along this same value. If one of the new training data splits has no elements, a leaf node is labeled with the most common value of t in S. This step is done to accommodate future data sets; while the training set may not have had any examples with that attribute value, other sets might and thus the tree assumes the

most common value is correct. All other splits of S are recursively passed to ID3() in line 45, along with t and the AttributeSet A (sans the attribute just used).

Suppose someone wanted to use the data set in Figure 3.1 to predict ticket sales. We have the TargetAttribute *Tickets*, an AttributeSet {*Month, Weekend*}, and a TrainingSet = {(3, 25, 0), (8, 45, 1), (8, 25, 0)}. After creating a root node, we first check if all values of the TargetAttribute *Tickets* are the same in the data set. This is not the case (two instances of 25 tickets sold, one instance of 45 tickets sold), so we continue through the algorithm. The second step is to check if AttributeSet is the empty set; this also is not the case. Thus, we enter the meat of the algorithm. We choose an attribute from AttributeSet and find the information gain of that attribute. Recall that the information gain of an attribute A and set of examples S is equal to $\text{Entropy}(S) - \sum_{v \in Values(A)} \frac{||S_v||}{||S||} \text{Entropy}(S_v)$. As established previously, the entropy of our TrainingSet is approximately 0.9235. With three entries, $||S|| = 3$. To start, let A be the attribute *Month*, which has possible values 3 and 8. For $v = 3$, the subset S_v has one entry, so $||S_v|| = 1$. In a set with only one entry, the entropy will necessarily be 0. For $v = 8$, S_v has two entries and $||S_v|| = 2$. The two August entries have different values for *Tickets*, so the proportion p_i will be $\frac{1}{2}$. Thus, the entropy of S_v for $v = 8$ will be $-.5\log_2(.5)$, which is .5. Thus, the information gain for attribute *Month* is:

$$\begin{aligned} .9235 - [(\frac{1}{3} * 0) + (\frac{2}{3} * .5] \\ .9235 - [0 + \frac{1}{3}] \\ .5902 \end{aligned}$$

Next, we repeat this formula for attribute *Weekend*. This attribute also has two possible values, 1 and 0. For $v = 1$, there is only one entry; $||S_v||$ will again be 1 and entropy will again be zero. For $v = 0$, there are two entries, so $||S_v|| = 2$. Both of

these entries have the same value for the target attribute *Tickets*, so the entropy on S_v for $v = 0$ is also 0. Thus, the information gain for attribute *Weekend* is:

$$.9235 - [(\frac{1}{3} * 0) + (\frac{2}{3} * 0)]$$
$$.9235 - [0 + 0]$$
$$.9235$$

We find that the *Weekend* attribute has a higher information gain than the *Month* attribute. So, the next step in the algorithm is to label the root node with *Weekend*.

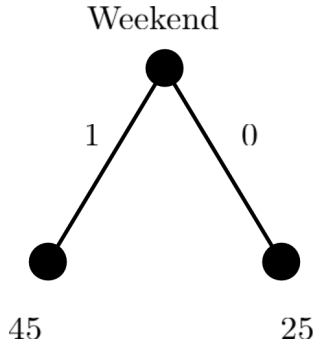

Figure 3.2: A decision tree made with the ID3 algorithm, based on the training set in Figure 3.1 with TargetAttribute *Tickets*

Then, for the two possible values of *Weekend*, the algorithm creates a new branch for that value. For the branch where $v = 1$, the ID3 algorithm is called with TargetAttribute *Tickets*, AttributeSet {*Month*}, and TrainingSet = {(8, 45, 1)}. In this recursive call, the TargetAttribute *Tickets* is uniform. Thus, the recursive call returns a node labeled with the value of *Tickets*, 45. For the branch where $v = 0$, the ID3 algorithm is called with TargetAttribute *Tickets*, AttributeSet {*Month*}, and TrainingSet = {(3, 25, 0), (8, 25, 0)}. Again, the TargetAttribute *Tickets* is uniform, so a node labeled with the value of *Tickets*, 25, is returned. Thus, the ID3 algorithm returns the tree in Figure 3.2.

Once the tree is generated by the ID3 algorithm, classifying new data is simple.

Suppose a new data set point is used, with *Weekend* equal to True (or 1) and *Month* equal to 5. Using the example in Figure 3.2, this event would be predicted to have 45 tickets sold.

3.1.2 Special Cases

While the data in Figure 3.1 use numeric data, the small sample set allows the ID3 algorithm to treat them as discrete values. However, decision trees must also be able to classify along continuous-valued attributes.

Temperature	Played Tennis
40	No
48	No
60	Yes
72	Yes
80	Yes
90	No

Figure 3.3: A sample data set containing the temperature in degrees Fahrenheit and whether or not tennis was played [6].

Example 3.1.1. Take for example the data set in Figure 3.3. This data set is being analyzed to determine which temperatures are ideal for playing tennis. For six days, the temperature in degrees Fahrenheit and whether people played tennis was recorded.

Since only six temperatures were recorded, this training set cannot use these findings as discrete values. If it was classified discretely, then any temperature not explicitly in the training set (68 degrees, for instance) would not be accurately classified. Thus, the temperature must be treated as a continuous-valued attribute.

To deal with a continuous-valued attribute A, a new Boolean attribute A_c is created where A_c is true if $A < c$ and false otherwise [6]. For the dataset in Figure 3.3, c should be set to a value where the target attribute (in this example, whether people played tennis) changes. Thus, for this example, the thresholds between whether people do play tennis or when they don't happens between 48 and 60 degrees, and between 80 and 90 degrees. The simplest method is to take the average of the two temperatures, so $A_{c1} = \frac{48+60}{2} = 54$ and $A_{c2} = \frac{80+90}{2} = 85$. Therefore, based on the training data, a decision tree classifying tennis by outdoor temperature would predict "No" when temperature is below 54 degrees or above 85 degrees, and "Yes" when the temperature is between 54 and 85 degrees.

Another hurdle with decision trees comes when a data set has missing values for particular attributes A. There are two strategies for dealing with these cases: assign the element the most common value of A in the rest of the data set, or assign probabilities for each possible value of A [6].

3.1.3 Optimizing the Tree

Decision trees are susceptible to overfitting, where decisions at each node are too closely tied to the training set. Overfitting is caused by an algorithm creating superfluous branches that slightly decrease entropy for the set, but are in fact arbitrary decisions [14]. For example, a decision tree could be created to classify students to their fields of study. An overfitted tree might contain choice nodes checking the names of the students. The tree is able to reach 0 entropy on the training set by classifying all students named "John Smith" as an English major and all students named "Jane Doe" as a Chemistry major, but this tree would not correctly classify a music major named John Smith.

There are two common methods to prevent overfitting. The first method is to require a minimum reduction in entropy at each split. Once the minimum is not

reached, the tree stops creating additional branches. While commonly used, this method is insufficient in cases where early splits do not reduce entropy, but later splits greatly reduce the entropy of the set [14]. The second method is to build the tree, then prune any extraneous nodes. Working from the bottom-up, the tree examines two leaf nodes with the same parent and determines whether merging these nodes does not increase entropy more than a specified threshold. If the nodes pass this test, they are combined and their parent becomes a new leaf node.

3.2 Conclusion

Decision trees are a useful tool to evaluate a data set, determine correlations between attributes and a target value, and create a tree to classify new data on these target values. The ID3 algorithm, one of the more common methods of creating a decision tree, examines each possible split of attributes to find the greatest reduction in the dataset's entropy, then creates a new branch on the tree using the attribute split. This process continues until all attributes are used or all data points have the same target value. The tree can be optimized by requiring a split in the tree to have some level of reduction in entropy. Once a tree is created, new data is classified by starting at the top of the decision tree, then following the branches until a target value is given.

CHAPTER 4

Description of Software

4.1 AI Model

4.1.1 Simulation Model

For this thesis, a 2D, turn-based role-playing game (RPG) was created, titled *Cave Escape*. In an RPG, players are given a selection of possible actions that their in-game character can take. These actions consist of, but are not limited to: attacking (usually with a melee weapon), using magical spells (of both offensive and defensive varieties), and using items (to heal players or otherwise make them more powerful). These choices are tied to the character the player controls, along with their character stats. A thief character may be faster and more nimble, but may also be weaker than other characters. A knight character could be stronger and better defended with their armor, but also slower and easier for opponents to hit. With regards to player actions and choices, the thief may be able to steal items, while the knight may be able to block attacks with a shield. This is the "role-playing" aspect of the game: the player chooses their actions based on the role of their in-game character.

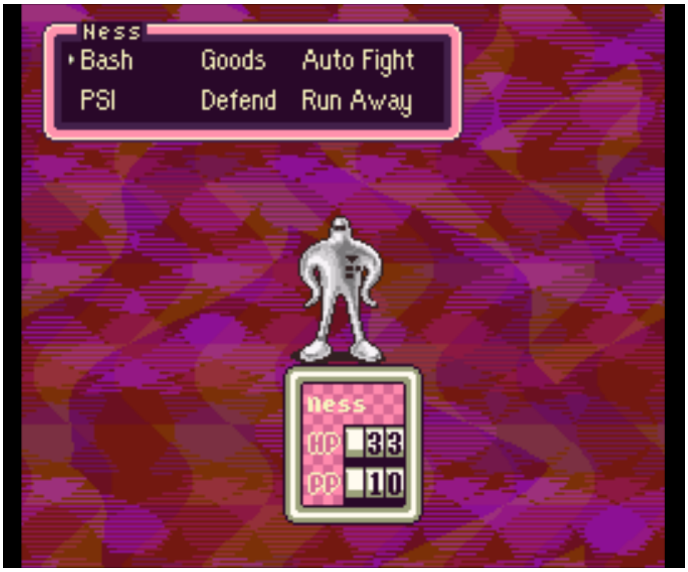

Figure 4.1: A screenshot of a battle from the turn-based role-playing game *Earthbound* [3].

For instance, observe the battle shown in Figure 4.1 from the game *Earthbound*. The computer opponent is shown in the center of the screen. At the top, the actions available to the player are listed. Players have the choice of making a melee attack (Bash), casting a magical spell (PSI), using an item (Goods), or reducing damage taken (Defend). Players can also select Auto Fight to have the computer make all the choices for them, or they can Run Away and escape from the battle. At the bottom, the hit points (HP) and psychic points (PP) of the character are shown - these correspond to the character's health and magical abilities, respectively. Specific to turn-based RPGs, the player has as much time as they want to decide on their strategy. This is in contrast to an action RPG, where combat takes place in real-time. Generally, action RPGs use 3D environments to better accommodate the real-time battles, but this is not always the case.

In the game created for this thesis, a turn-based approach is better suited because it emphasizes player strategy. Action RPGs may have similar lists of commands and actions, but these actions frequently have a corresponding skill component. Take for example a basic attack. In a turn-based RPG, this action requires little from

the player: they select that choice from a list, and their in-game character attacks the opposing player. Depending on the game, this attack may or may not be successful - the opponent could dodge or block the hit - but the success or failure is determined by the game through the stats of both players. For instance, a character may have a high enough agility to dodge an attack, or a high enough defensive to block the strike; the role-playing aspect of turn-based RPGs often boils down to how players build their character's stats. An action RPG, on the other hand, determines whether an attack misses or hits through hit-boxes. A hit-box is a hidden area tracked by the game code and tied to some in-game object; when hit-boxes intersect, something happens depending on the types of objects. For this example, an attack would be successful if the hit-box of the character's attack (their sword, fist, magic spell, etc.) intersects with the hit-box of their opponent's body. The attack would fail if these hit-boxes do not intersect. Thus, the overall success of a character in an action RPG is dependent on a player's skill in making hit-boxes collide. This type of gameplay does not lend itself well to game theory analysis, as a player could make all the right decisions but still lose from lack of skill.

Figure 4.2: A screenshot of a battle from the action role-playing game *Xenoblade Chronicles* [7].

For a more concrete example, regard a battle from the action RPG *Xenoblade Chronicles*, as shown in Figure 4.2. In this game, the player can move freely in 3D

space. The character shown currently has four abilities, represented as icons along the bottom of the screen. The two red-tinted abilities are attacks, as is the shaded center ability. The blue ability on the right is a healing spell. The skill component of this game comes with the two red abilities; these abilities do more damage and cause additional effects when performed at certain angles. For instance, the leftmost ability reduces an opponent's defense when used on the enemy's side, while the second red ability does more damage when used from an opponent's back. Thus, two players could use the same series of abilities but have different outcomes based on how their attacks were positioned.

Similarly, the game presented in this thesis is a two-dimensional game. With a turn-based RPG, since the gameplay involves choosing actions from a list, the only difference between a 2D and a 3D turn-based RPG is the graphical quality. A 2D game uses sprites, while 3D games use polygonal models. A two-dimensional game was chosen to simplify development.

To begin building an AI model, we first create a simulation of the *Cave Escape*. The simulation has two players, each with 25 health points (HP). In this first simulation, both players have two possible actions per turn which are randomly selected: attack the other player, reducing their opponents HP by 5 points, or heal themselves, increasing their HP by 4 points. Players can not heal above their starting value of 25 HP; any extra health points are discarded. Each action has a 50% chance of occurring. The game runs until one player's HP is reduced to 0. The total number of turns and the sequence of actions are recorded into a CSV file. Each line of the CSV file contains the length of the game, a sequence of Player 1's HP values, then a sequence of Player 2's HP values. After running 4500 simulations, the data is imported into MATLAB. A subset of the games are used as a training set for a decision tree, which predicts the player that wins the game.

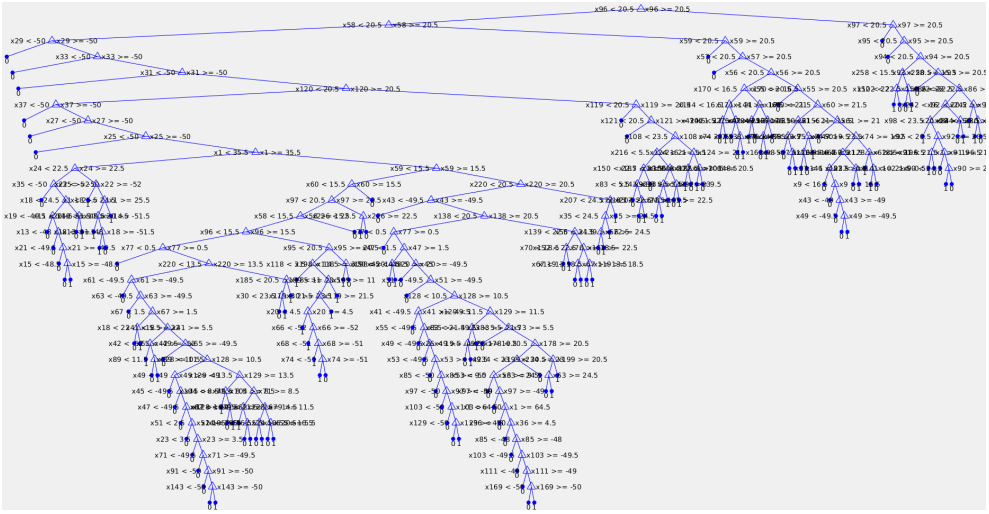

Figure 4.3: The decision tree generated from 4000 simulations of the game. The predicted attribute is a Boolean value of 0 when player 1 wins and 1 when player 2 wins.

However, this method is inconclusive, for a number of reasons. Firstly, the decision tree includes a large number of branches, as seen in Figure 4.3. Secondly, the actual criteria evaluated at each branch of the tree is different for each game. Aside from the first column, which contains the total number of turns in the game, and the second column, which contains the starting HP of player 1, the values in any particular column differ from game to game. For instance, the root node uses the value in column 96 as its classifier. In one game, column 96 may contain a HP value for Player 2. In another, longer game, that column may contain a HP value for Player 1. Thirdly, the individual games contain numerous streaks: a longer game is not the result of better strategy, but is instead caused by streaks where both players choose to heal on their turns. Players in these streaks reach HP levels near the 25 HP cap, effectively resetting the game.

The lack of consistency along columns is most likely the largest detriment to this model. Decision trees work best when classifying problems on specific attributes; if the attribute is different at column 96 from one game to another, the information gain at that column is meaningless. This is likely behind the sprawling nature of the

decision tree itself. Without clear attributes to test, each split only has minuscule amounts of information gain.

4.1.2 Extensive Tree Model

A game theoretic angle is used on the second attempt. The HP of both players is reduced to 10, and two restrictions are added on healing: a player can not heal when they have full health, and a player can only heal themselves twice. As in the first attempt, players can only attack or heal. With these new rules in place, we create an extensive-form tree for the new game.

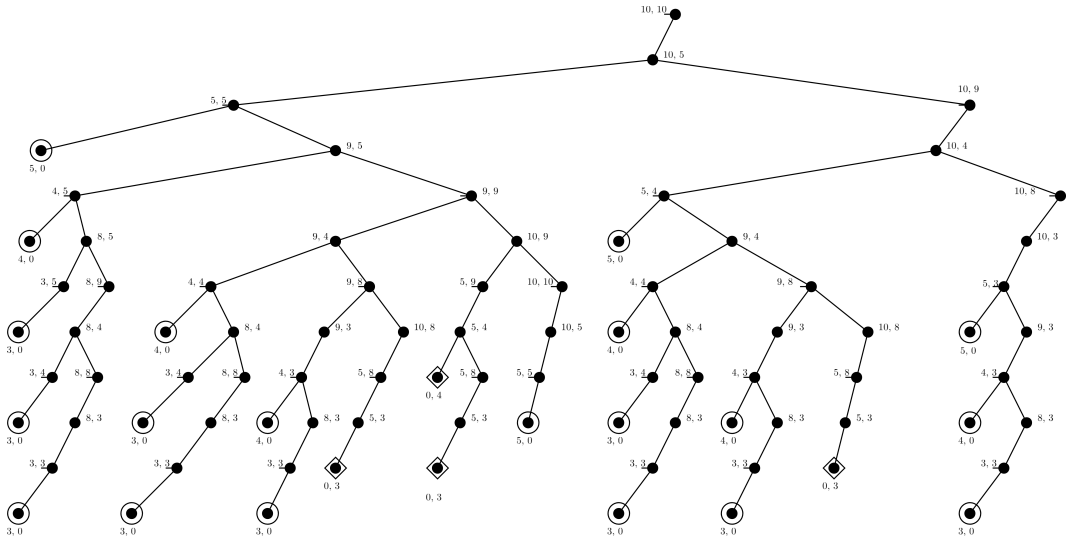

Figure 4.4: The extensive-form tree for a simplified game. Each player starts with 10 hit points (HP). At each node, a player can choose to heal themselves or attack their opponent. Attacking reduces the opponent's HP by 5 points. Healing increases the decision maker's HP by 4 points; they can only heal themselves twice per game.

In Figure 4.4, Player 1's nodes are denoted by a small dash to the left of the node. The HP values of both players are recorded at each node. Leaf nodes where Player 1 wins are enclosed in a circle, and leaf nodes where Player 2 wins are enclosed in a diamond. There are 24 outcomes for this game; of these, only 4 of them result in a win for Player 2, while the other 20 result in wins for Player 1. In fact, with optimal play, Player 1 always wins in this game.

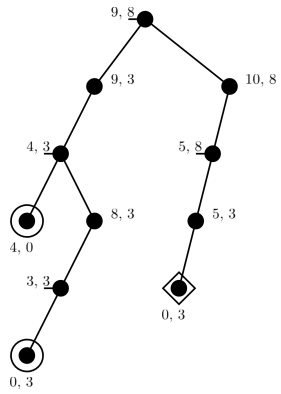

Figure 4.5: A subtree of the extensive-form representation in Figure 4.4. This subtree contains one of player 2's only winning outcomes.

For example, in the subtree shown in Figure 4.5, both players have a winning outcome. However, the critical decision is made by Player 1. If Player 1 chooses to heal at the root of this subtree, then Player 2 will win in 3 more turns. If Player 1 chooses to attack, then Player 1 will eventually win. Thus, in any game which reaches this subtree, Player 1 will choose to attack on their turn since that choice leads to their victory.

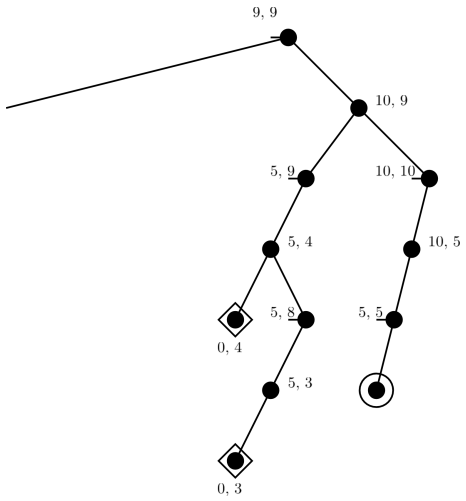

Figure 4.6: A second subtree of the extensive-form representation in Figure 4.4. This subtree contains two more of player 2's winning outcomes.

Two more of Player 2's winning outcomes can be seen in the subtree in Figure 4.6. At the root node of this subtree, Player 1 is making their choice. If they choose to

heal, the game moves into the right child. Player 2's first choice on the right child's subtree either results in a win (if Player 2 attacks) or a loss (if Player 2 heals). Thus, an intelligent Player 2 would always attack at this stage, since it is a guaranteed win. However, if Player 1 knows that this is a guaranteed win for their opponent, Player 1 can avoid this threat by attacking instead of healing at the root node.

If Player 1 decides to avoid the subtree in Figure 4.6, Player 2 has the next choice. If Player 2 decides to heal at this new node, then the resulting subtree is identical to the one found in Figure 4.5. This subtree contains Player 2's final winning outcome, but once again Player 1 can avoid this outcome with ease.

4.1.2.1 Backwards Induction on Extensive Tree Model

We can prove that Player 1 will always win this game through the backwards induction algorithm discussed in Chapter 2. Recall that the algorithm travels through a game tree by recursion and, at each node, returns the child node which gives the player acting at the parent node the most utility.

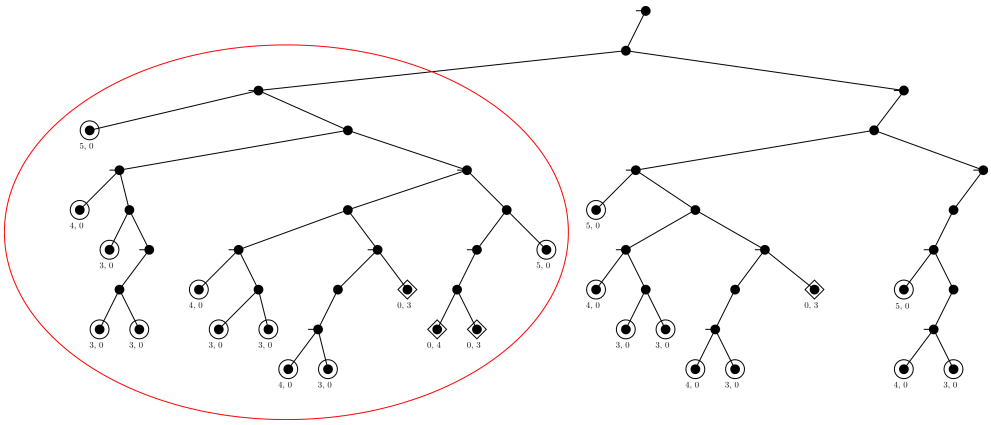

Figure 4.7: The extensive-form tree for the simplified version of the game, but with extraneous nodes removed; a node is extraneous if the player making their decision has only one possible action. The circled subtree will be the first subtree examined on recursive calls of the backwards induction algorithm.

For the example in Figure 4.4, we can trivially condense branches where only one action is available, taking the utility values from the leaf nodes and copying

them up the tree. This produces the tree seen in Figure 4.7. We define the utility of a player in this game as their HP at the end of the game minus the HP of their opponent. Thus, for a game with the outcome (4, 0), Player 1 will have a utility of 4 and Player 2 will have a utility of -4.

The algorithm begins at the root node of the tree. Since the root is not a leaf node, the algorithm is called on the root's children, starting with the leftmost child. The root node only has one child, since players cannot heal when their HP is maxed out. So, the backwards induction algorithm is called on the next node, which does have more than one child. To find the utility of both children, the algorithm must be recursively called on both subtrees, starting with the subtree circled in red in Figure 4.7.

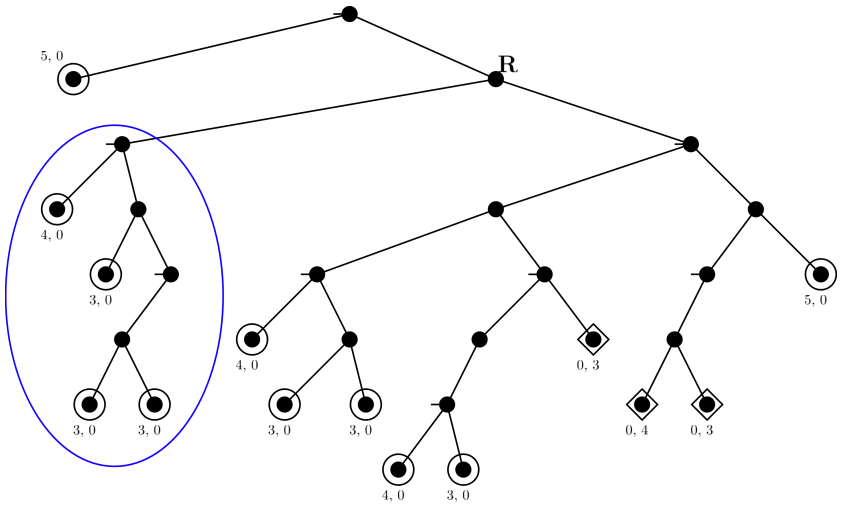

Figure 4.8: The tree examined by the backwards induction algorithm after recursing to the red circled subtree in Figure 4.7. The blue circled subtree is the next recursive step of the algorithm.

Examining the root of the subtree in Figure 4.8, we find that the left child of the root is a leaf node. Thus, since the root is a decision node for Player 1, the utility of this entire subtree will be the maximum utility for Player 1 between the leaf node (5, 0) and the utility of the right child, which is its own subtree. Once again, the

backwards induction algorithm is called recursively, this time on the subtree with root *R*. On the recursive call, the left child of *R* is the subtree circled in blue.

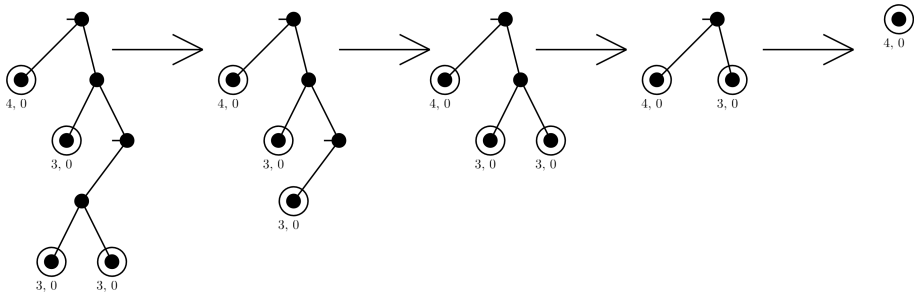

Figure 4.9: The steps taken by the backwards induction algorithm after reaching the circled subtree in Figure 4.8.

As the algorithm continues to recurse through the game tree, it will eventually reach the pair of leaf nodes at the bottom of the leftmost tree in Figure 4.9. The preceding decision node is a choice for Player 2, so Player 2 will make whichever choice gives them more utility. In this example, both leaf nodes provide Player 2 with -3 utility, so the utility of the decision node is (3, 0). This utility value is moved up the tree to the next decision node, as shown in Figure 4.9. These steps continue up the subtree. The next decision node again has a tie between (3, 0) and (3, 0), but the root of the subtree has a choice between (3, 0) and (4, 0). Since the root of this subtree is a decision node for Player 1, that player will prefer to attack and reach the (4, 0) leaf node for 4 utility. Thus, the utility of the circled subtree in Figure 4.8 is (4, 0).

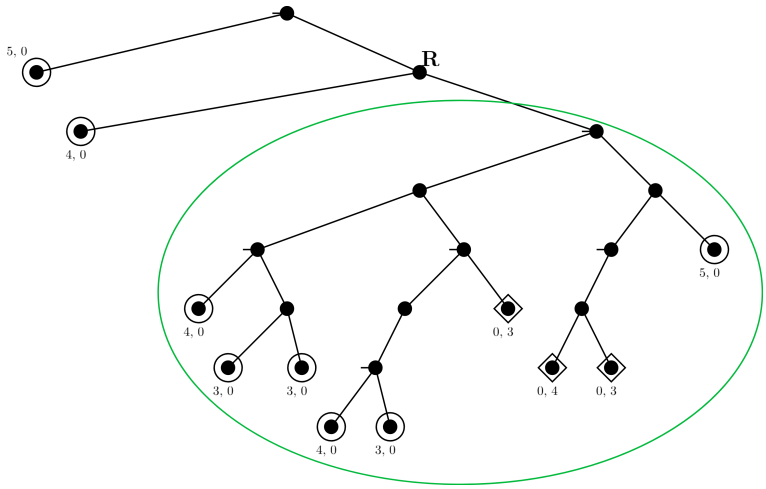

Figure 4.10: The subtree in Figure 4.8, after the backwards induction steps performed in Figure 4.9.

Thus, after exploring the subtree in Figure 4.9, the left child of node *R* is changed from a subtree to a single leaf node, the leaf with utility (4, 0). Now that the left child of *R* is a leaf node, the algorithm continues along the right subtree of *R*, circled in green.

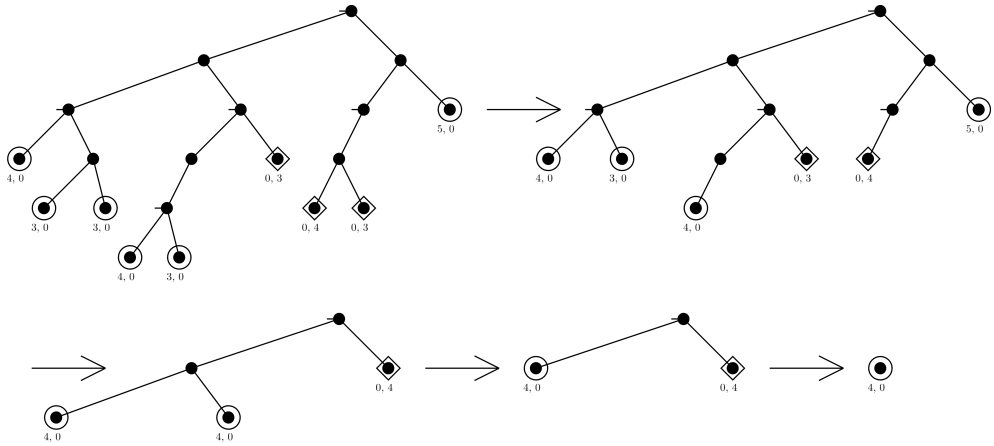

Figure 4.11: The steps taken by the backwards induction algorithm after reaching the circled subtree in Figure 4.10.

In the new subtree, the induction algorithm again determines which of the leaf nodes to bring up the tree. The parent of the leaf nodes decides which leaf provides the most utility for that player. In the first step, along the bottom of the tree and

from the left, Player 2 makes a choice between (3, 0) and (3, 0); Player 1 makes a choice between (4,0) and (3, 0); Player 2 makes a choice between (0, 4) and (0, 3). Respectively, the algorithm chooses (3, 0), (4, 0), and (0, 4). Each of these best leaf nodes are moved up the tree, resulting in the second tree in Figure 4.11. From here, extraneous branches are removed and three more decisions are made: player 1 chooses between (4, 0) and (3, 0); Player 1 chooses between (4, 0) and (0, 3); Player 2 chooses between (0, 4) and (5, 0). Respectively, the new utilities of the decision nodes are (4, 0), (4, 0), and (0, 4), creating the third tree in Figure 4.11. The next decision is made by Player 2, and is a tie between the two (4, 0) nodes. Then, in the fourth tree shown in Figure 4.11, Player 1 makes their choice between the (4, 0) and (0, 4) nodes. Since Player 1 wins on the (4, 0) node, this is the utility of the entire subtree.

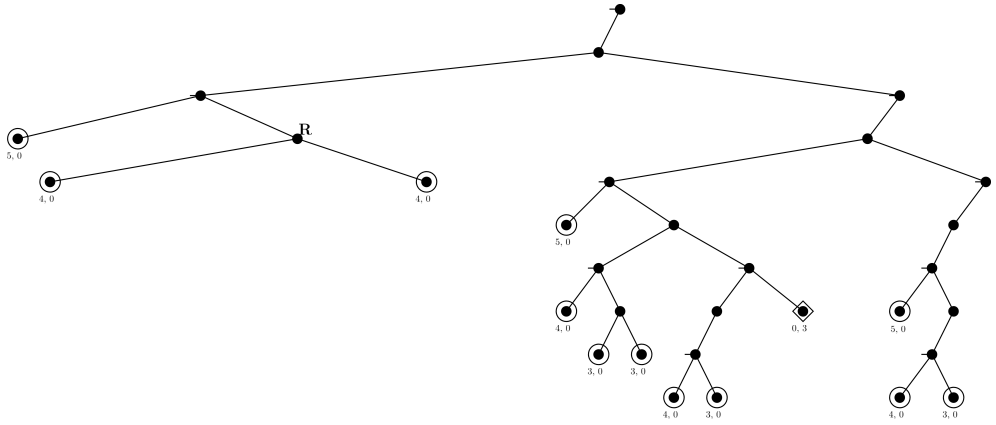

Figure 4.12: The full game tree, after the subtrees are evaluated by the backwards induction algorithm in Figure 4.9 and Figure 4.11.

Now that most of the left side of the tree has been explored by the backwards induction algorithm, we return to view the full tree in Figure 4.12. The node *R*, a decision node for Player 2, must choose between two identical leaf nodes, both with (4, 0) utility. Thus, *R* given a utility value of (4, 0). For the final decision on the left subtree, Player 1 chooses between a (5, 0) leaf node and a (4, 0) leaf node, and will

select the former. Thus, the entire left subtree of is equivalent to a leaf node with utility (5, 0).

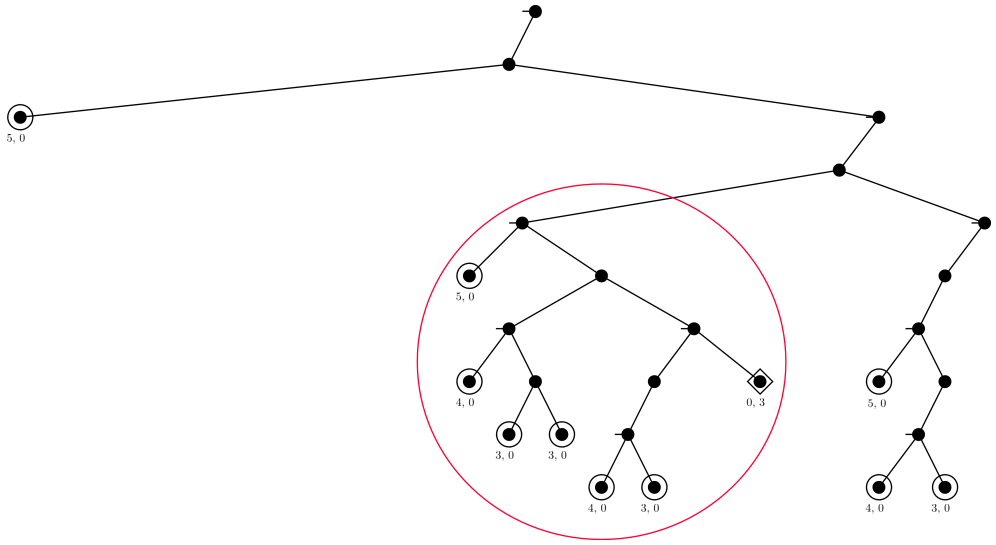

Figure 4.13: The full game tree, after the backwards induction algorithm has recursed through half of the tree. The next subtree to be examined with the algorithm is circled in red.

Now that the left subtree is reduced to a single leaf node, the algorithm moves to the right subtree and continues working. The next tree examined will be the circled subtree in Figure 4.13.

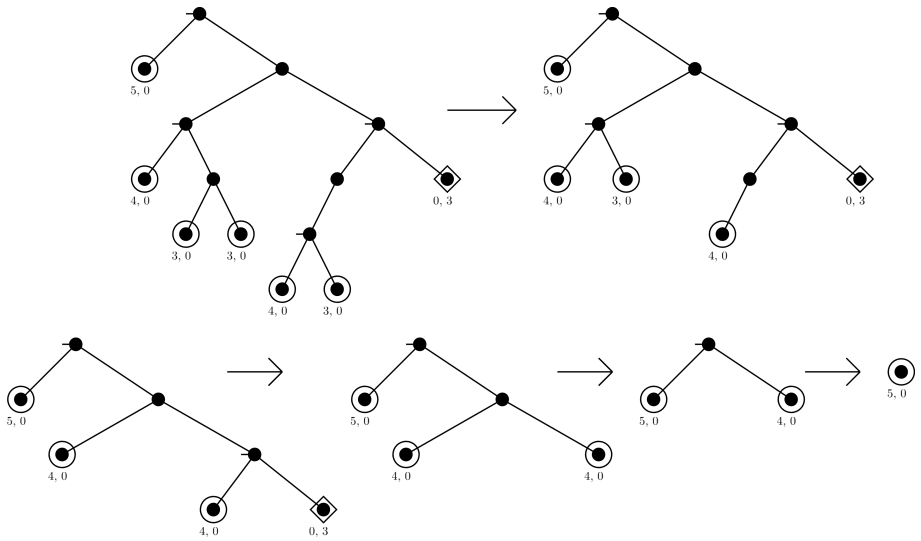

Figure 4.14: The steps taken by the backwards induction algorithm after reaching the circled subtree in Figure 4.13.

In the new subtree, the induction algorithm again determines which of the leaf nodes to bring up the tree. In the first step, at the lowest levels of the tree, player 1 makes a choice between (4, 0) and (3, 0), and player 2 makes a choice between (3, 0) and (3, 0). Player 1 chooses the former and player 2 chooses either. In the second step, player 1 chooses between (4, 0) and (3, 0) again, choosing the (4, 0) node again. In the third step, player 1 chooses between the (4, 0) and (0, 3) leaf nodes; player 1 will select the (4, 0) since they win at that node. In the next two steps, player 2 chooses between two equivalent (4, 0) nodes, then player 1 chooses between a (4, 0) node and a (5, 0) node. Thus, the subtree is equivalent to the (5, 0) leaf node.

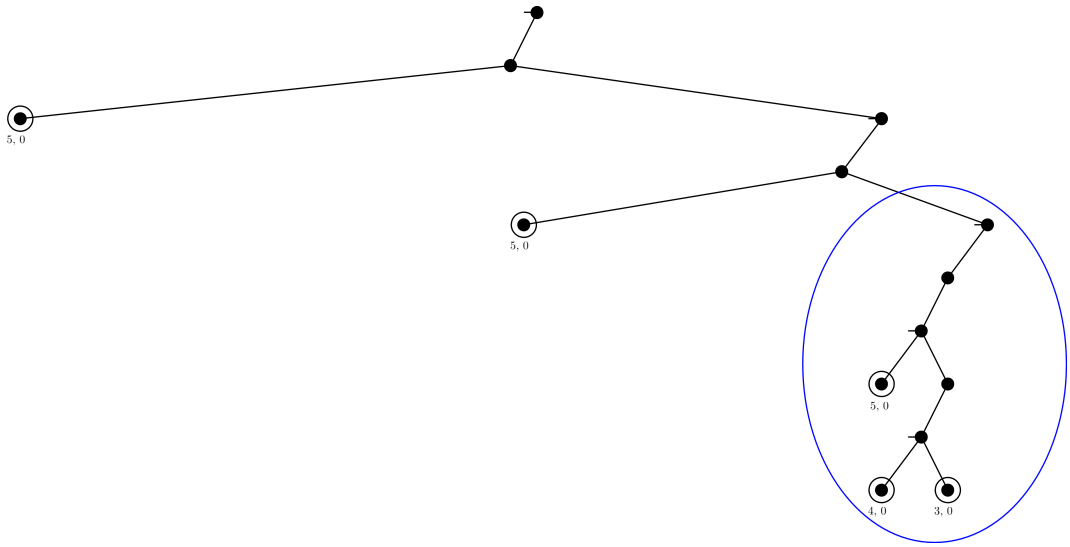

Figure 4.15: The game tree in Figure 4.13, after the backwards induction steps performed in Figure 4.14. The blue circled subtree is the next to be evaluated by the algorithm.

With the subtree in Figure 4.14 evaluated, the subtree is replaced with the leaf node (5, 0) as shown in Figure 4.15. Next, the algorithm will perform the same analysis on the subtree circled in blue.

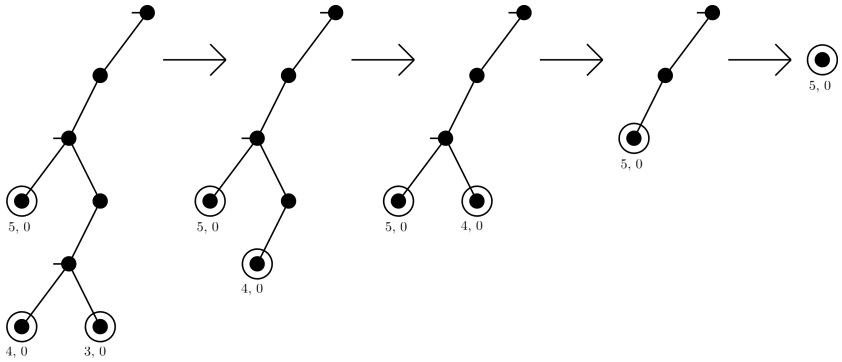

Figure 4.16: The steps taken by the backwards induction algorithm after reaching the circled subtree in Figure 4.15.

For this final subtree in Figure 4.16, player 1 first makes a choice between the (4, 0) and (3, 0) leaf nodes at the bottom of the tree and picks the former. Since player 2 only has one choice at their decision node, the (4, 0) node is brought up one level more. Player 1 makes a second choice between (4, 0) and (5, 0), choosing the latter. There are no more decisions on this section of the tree, so the subtree is equivalent to a leaf node with utility (5, 0).

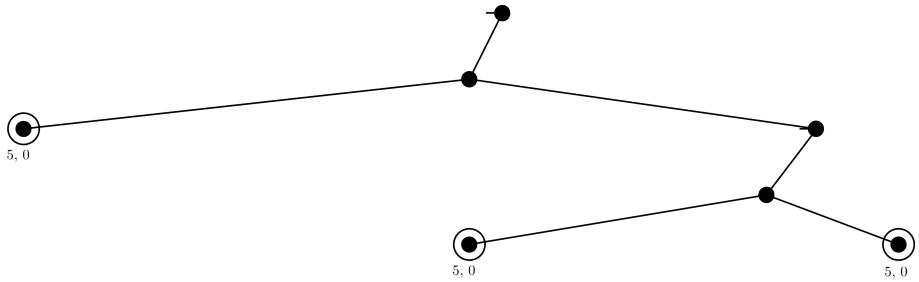

Figure 4.17: The game tree in Figure 4.15, after the backwards induction steps performed in Figure 4.16. At this point, all leaf nodes have equal utility, so the entire tree is equivalent to the single leaf node with utility (5, 0).

After the steps taken in Figure 4.16, the tree for the entire game is equivalent to the tree in Figure 4.17. At this stage, all the leaf nodes have the same utility value: (5, 0). Thus, no matter which player is making the decision, any choice they make in this tree will lead to an outcome of (5, 0). If both players are making optimal decisions, the outcome of a game in Figure 4.17 will therefore always be (5, 0).

4.1.2.2 Analysis

Player 1 has a huge advantage in this game by taking the first action. If both players only choose to attack, then Player 1 wins the game. The utility of this outcome is (5, 0), and is thus one of the optimal strategies for the game, as established. In fact, Player 1 can only lose by choosing to heal at certain points in the game.

In all four paths where Player 2 won, there is a node where Player 1 chose to heal from 9 HP to 10 HP. If Player 1 only attacks, then Player 2 has no chance of victory. Since the normal amount of HP recovered by healing is 4 HP, we can see that Player 1 is wasting potential HP by using one of their turns to heal when their HP is already close to the HP cap. We may infer that, in games with larger HP caps, the same effects would appear. It is much more useful for a player to heal when their HP is closer to zero.

In this limited model, it is clear that healing is not a viable strategy. Since players can only attack or heal, one player's choice to heal will either have no effect on their utility or a negative effect on their utility. If Player 2 heals and Player 1 immediately attacks, then Player 2 has a net loss of 1 HP. The HP recovered by a player is not enough to offset the damage dealt by their opponent. If a player heals and their opponent also heals, then the difference in utility is zero. Situations where both players heal at the same time does not change the overall performance of the players. If Player 2 was losing to Player 1, and both players heal, then Player 2 will still be losing to Player 1. At the most, choosing to heal only delays the inevitable.

Now that some patterns in the game are identified, we expand the choices of the game and build a prototype using Python and the pygame library.

4.2 Game Engine

Python was chosen as the development language for this game. Python is an extremely portable language: anyone with Python installed can run the code. Python also has a robust library for two-dimensional games called pygame, which was essential in creating this game. Specifically, Python 3 was used for better compatibility with the pygame library and for the `copy()` function for arrays.

4.2.1 The pygame Library

The pygame library provides a number of Python functions that are useful in creating a two-dimensional video game. The library was originally created in 2001 to combine Python with SDL (Simple DirectMedia Layer), a library of multimedia controls written for C [16]. pygame can utilize a variety of different graphics libraries, including OpenGL, DirectX, the Linux frame buffer, and an ASCII art backend [16]. The pygame library is supported by numerous operating systems, and the core functions of pygame use highly optimized C or assembly code.

At the top level, pygame controls the initialization and exiting of its various modules, particularly the `pygame.display` module which renders the game. pygame features two functions to update a display window, `pygame.display.flip()` and `pygame.display.update()`. `display.flip()` works with two separate arrays, containing the pixel data for the window in which the game operates. One array is displayed on the screen, while the other array is used to record changes made to the on-screen image. Once all changes are made, `display.flip()` swaps these arrays, or "buffers," by copying all the data from one array into the other. The buffer that recorded changes is now displayed on-screen, while the buffer that was displayed can now be used to record further changes. `display.update()` is an optimized version of `flip()` that takes as an argument a rectangle or a sequence of rectangles.

These rectangles correspond to the areas of the display that need to be updated. For instance, passing the coordinates of a rectangle over a game's scoreboard only updates the scoreboard. With this function, large sections of the pixel array do not need to be copied from one buffer to another, speeding up the rendering process.

The two main objects in pygame are the `Surface` object and the `sprite` object. Surfaces can be changed with pygame to alter various attributes. For instance, the `alpha` value, or the transparency of the surface, can be changed with `set_alpha()`. The individual color values can be converted to integers and vice versa with the `map_rgb()` and `unmap_rgb()` functions, respectively. This allows for pygame to store colors as a single number, rather than a tuple of integers. Surfaces are mostly used to load image files and to create backgrounds for the game. Sprites, on the other hand, are used for the actual in-game objects, such as enemies, player characters, and projectiles. Sprites can be stored in a `Group` object, that can be used to separate sprites by different purposes. Each sprite draws its image to a `Surface`, provided that it has a `Surface.image` and `Surface.rect` attribute.

Once a sprite or surface is created with an image, pygame can also perform graphical transformations on that image. The `pygame.transform` module has functions to flip, rotate, and scale an image. Another pygame feature is the ability to do collision detection on sprites and rectangles.

For the actual gameplay of a pygame program, there are also modules for joystick, mouse, and keyboard controls. A sound mixer module allows audio tracks to be played in the game, and the `pygame.time` module can be used to control the framerate of the game.

4.2.2 Gameplay Design

For the expanded version of the game, several things were changed from the game in Figure 4.4. Two additional choices were added: Parry and Strong Attack. With Parry, the player enters a guarded stance until their next turn; this player will be referred to as the defending player. If, before their next turn, their opponent attacks them, the defending player counter-attacks. The defending player loses no HP, and the player who attacked them loses 3 HP from the counter-attack. A Strong Attack does more damage than a regular attack (7 HP vs 5 HP), but has a chance to miss the opponent completely and do no damage. Additionally, a Strong Attack is unaffected by Parry; if a player is Parrying and their opponent uses a strong attack, the parry is unsuccessful and the defending player suffers 7 HP of damage.

```
1  class PlayerAvatar:
2    def __init__(self, model):
3      self.hp = 25
4      self.maxHP = 25
5      self.healTurns = 2
6      self.isParrying = False
7      self.aggressiveness = .25
8      self.AI_MODEL = model
9  # AI Helper Function - Picks the appropriate function to
       call, and passes the correct
10 # information
11   def helperAIFunc(self, enemyHP, isParrying):
12     if self.AI_MODEL == 1:
13       return self.randomAI()
14     elif self.AI_MODEL == 2:
15       return self.aggressiveRandomAI()
```

```
16      elif self.AI_MODEL == 3:
17          return self.fiftyPercentAI()
18      elif self.AI_MODEL == 4:
19          return self.comparativeAI(enemyHP)
20      elif self.AI_MODEL == 5:
21          return self.scalingDifficulty(enemyHP)
22      elif self.AI_MODEL == 6:
23          return "A"
24      elif self.AI_MODEL == 7:
25          return self.parryCountering(enemyHP, isParrying)
26      else:
27          return self.limitedRandomAI()
```

Listing 4.1: The `PlayerAvatar` class and its helper function, `helperAIFunc()`.

The two players in the game begin with 25 HP each, and are able to heal twice per game. These values are stored in a `PlayerAvatar` class, as shown in Listing 4.1 at lines 3-5. This class also stores the type of AI used for the computer player, at line 8. The class has a helper function, `helperAIFunc`, which is called from within the main game loop. In this helper function, the class variable `self.AI_MODEL` is used in a series of `if` statements to call that model's respective function. Notice that the `elif` statement at line 22 does not call a function, but instead returns the character "A." This corresponds to the baseline AI model, assigned to the Beserker character; this model and all the other AI functions will be discussed later in this section. Each of the AI functions in the helper function return one of four characters: "A," "S," "P," or "H." These correspond to the four possible actions in the game: Attack, Strong Attack, Parry, and Heal. These characters are returned to the main game loop, which adjusts the game based on the action taken.

Eight different AI models were created. In all models, the AI checks to see if

any healing turns remain; if not, the AI chooses a different option. The specific replacement action varies from one model to another. Three of the models rely on a random number generator to make their decisions.

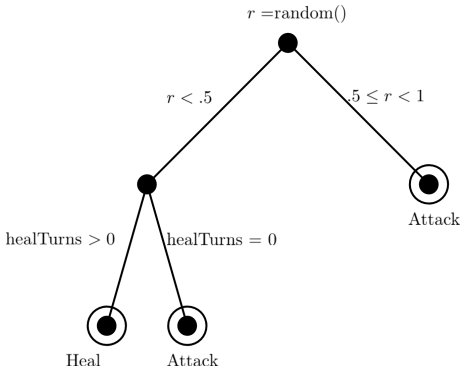

Figure 4.18: The decision tree for the `limitedRandomAI()` function.

The first AI model, the `limitedRandomAI()` function, chooses randomly to attack or heal with a 50/50 chance of either, but will use a normal attack instead if the AI has already healed itself twice. This is shown by the decision tree in Figure 4.18. The variable r is given a random value between 0 and 1, and the first decision in the tree is based on the value of r. After that, the tree has either reached a leaf node (for $r \geq .5$) or must make a second decision, this time based on the number of healing turns remaining. Given the limited success Player 2 had in the original game tree in Figure 4.4, this model was not used in the final version of the game. Instead, a different model was used which could perform all four possible actions.

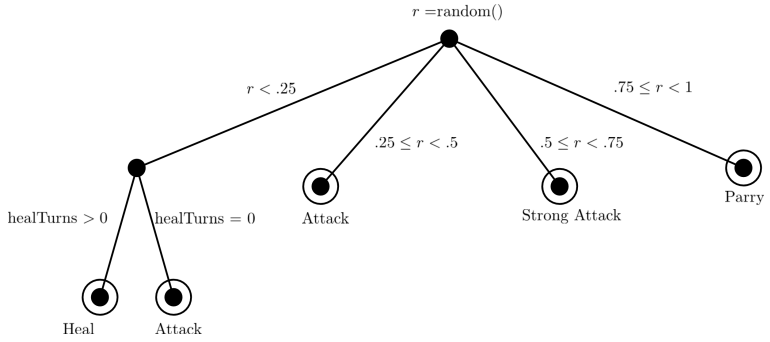

Figure 4.19: The decision tree for the `randomAI()` function.

The second model, `randomAI()`, has a 25% chance of attacking, healing, parrying, or using a strong attack. Again, Figure 4.19 shows that if the random number generator chooses to heal, but all healing turns have been used, the AI does a normal attack instead. This choice was made to affect a sense of desperation in the AI: when an opponent no longer can heal themselves, they become more offensive to try and win the battle. This kind of behavior - where the AI changes their strategy after all healing turns have been spent - is incorporated into all the models.

```
1  # AI Model 2 - Randomly choose between 4 actions: Attack,
       Strong Attack, Parry, and Heal.
2  # If no heal turns remain, the AI does a normal attack
3  def randomAI(self):
4    newAction = ""
5    r=random()
6    if(r<.25 and self.healTurns > 0):
7      newAction = "H"
8    elif(r<.5):
9      newAction = "A"
10   elif(r<.75):
11     newAction = "S"
12   else:
13     newAction = "P"
14   return newAction
```

Listing 4.2: The `randomAI()` function.

Listing 4.2 shows the code used to create the decision tree in Figure 4.19. In the code, each range of values is compared to the random variable r - Python's `random()` function returns a number in the range $[0, 1)$ - and the action is determined by the range into which r falls. Since Python executes each line of code sequentially,

lower bounds on the ranges are not needed; the lower bounds are implicit from the previous `if` or `elif` statements. In comparison with the tree, the program condenses the two Attack nodes together with the first `elif` clause. Since the first `if` statement checks for both `r<.25` and `self.healTurns>0`, the two branches of the decision tree leading to the Heal leaf node are both covered. This format cannot be done directly with decision trees, since each decision node in the tree must use a single variable to split its branches.

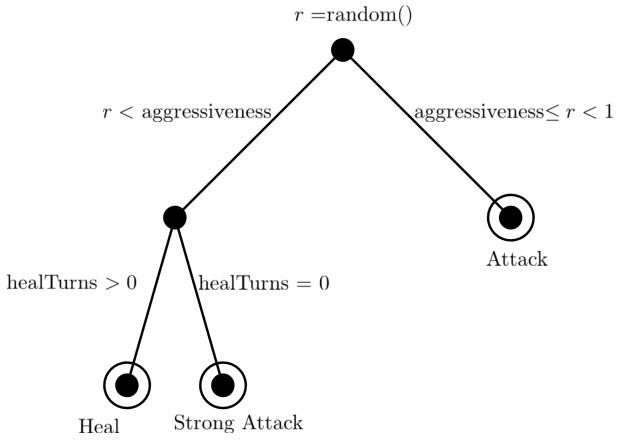

Figure 4.20: The decision tree for the `aggressiveRandomAI()` function.

The third model, `aggressiveRandomAI()`, is the final model which uses a random number generator. For this model, there is an additional variable given to the AI player, the aggressiveness stat. Rather than evenly splitting up actions 50-50, the choice between attacking and healing is split along this aggressiveness, which is a real number between 0 and 1. For this thesis, the aggressiveness was set to .25, leading to an AI which attacked 75% of the time and healed 25% of the time. As can be seen in the decision tree in Figure 4.20, the AI uses strong attacks when the random number r is below the aggressiveness stat and the two allotted healing turns are expended.

```
1 # AI Model 3 - Use agressiveness statistic to determine how
      frequently the AI attacks.
```

```
2 # If the AI is randomly chosen to heal, but has expended all
      healing turns, it uses a
3 # strong attack instead.
4 def aggressiveRandomAI(self):
5   newAction = ""
6   r=random()
7   if r<self.aggressiveness and self.healTurns > 0:
8     newAction = "H"
9   elif r<self.aggressiveness:
10    newAction = "S"
11  else:
12    newAction = "A"
13  return newAction
```

Listing 4.3: The `aggressiveRandomAI()` function.

Like in the `randomAI()` function, the `aggressiveRandomAI()` function in Listing 4.3 is able to condense branches and nodes with `elif` and `else` statements. The `aggressiveness` variable is an element of the `PlayerAvatar` class, and is set to .25.

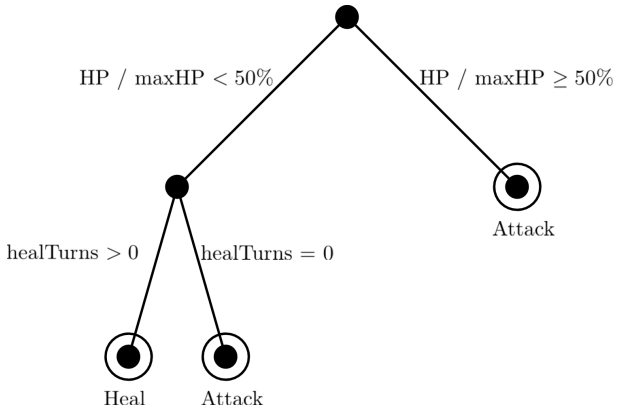

Figure 4.21: The decision tree for the `fiftyPercentAI()` function.

In the fourth model, `fiftyPercentAI()`, the AI only heals when its health falls

below 50% of its maximum. Thus, the AI only heals when it is close to death, as opposed to the random AI models which could potentially waste their heals at the start of a game. As shown in Figure 4.21, this AI only uses normal attacks and heals; it does not do any parrying or strong attacks. Thus, it has the most in common with the `limitedRandomAI()` function with regards to possible actions. However, by saving heals until later in the game, this model has some semblance of intelligence, whereas the `limitedRandomAI()` does not.

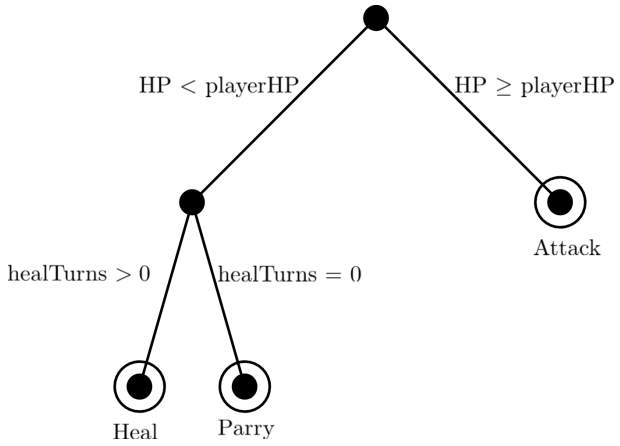

Figure 4.22: The decision tree for the `comparativeAI()` function.

The fifth model uses similar tactics as the fourth, but instead of using 50% as the threshold, the `comparativeAI()` function heals whenever its HP drops below the player's HP, as seen in Figure 4.22. Additionally, when the AI has expended all its healing turns and its HP falls below the player's HP, the AI uses a parry instead of an attack. Over the course of a game, the AI will defend itself, relying on counter-attacks to weaken the player before making attacks of its own.

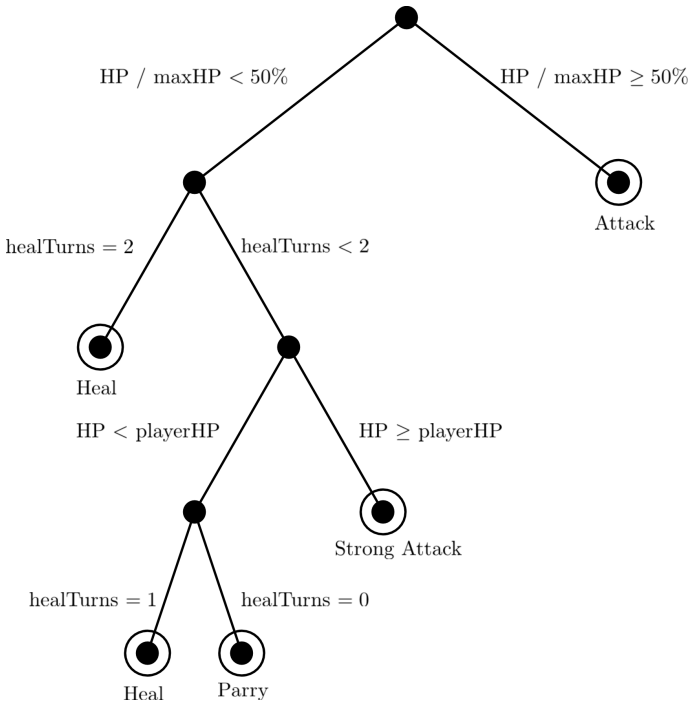

Figure 4.23: The decision tree for the `scalingDifficulty()` function.

The `scalingDifficulty()` model in Figure 4.23 is a loose combination of the `fiftyPercentAI()` and `comparativeAI()` functions. As in 4.21, the AI performs a normal attack whenever the AI's HP is above 50% of its maximum. In this model, the two available healing turns are used in different circumstances. The first healing turn is used when the AI drops below 50% HP for the first time. If their HP drops below 50% after this first healing turn, the AI becomes more strategic. If the AI is below 50% but still has a greater HP value than the player, then the AI will use strong attacks. However, if the AI is both below 50% and below the player's HP, then it will use its second healing turn if available and parry if no healing turns remain.

Figure 4.24: The decision tree for a baseline AI model, which only uses normal attacks.

After some preliminary testing, the final two models were created. The first of these is a baseline model, which only uses normal attacks. As Figure 4.24 depicts, the decision tree for this model is a single leaf node.

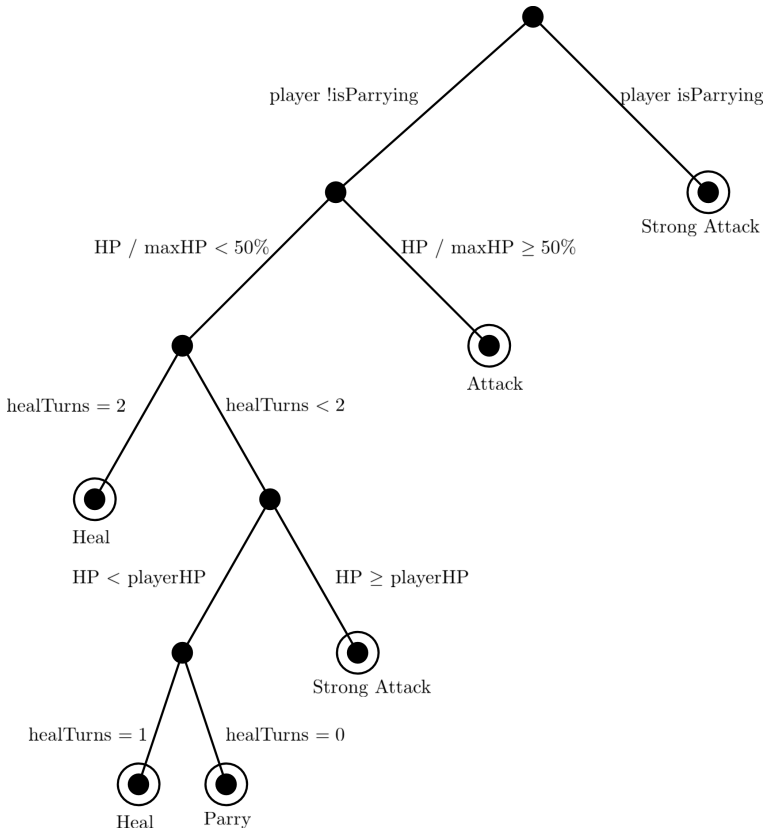

Figure 4.25: The decision tree for the `parryCountering()` function.

The second of these new models was designed to be more offensive and use strong attacks whenever the human player chose to parry. Otherwise, the second new model acts the same as the `scalingDifficulty` model.

4.2.3 Visual Design

The visual design of the game was left fairly simple, to avoid distracting players from the strategic elements of the game. Icons for the player character and enemy opponents were created in the GNU Image Manipulation Program (GIMP), then

imported as pygame sprites using the `pygame.image.load()` function. Since these sprites had a transparency component, the function `convert_alpha()` was also used to preserve this transparency in pygame.

Figure 4.26: The set of sprites used for the characters in *Cave Escape*. Clockwise from top left: the player character, an orc, an elf, a beserker, a knight, a magician, a troll, and a goblin.

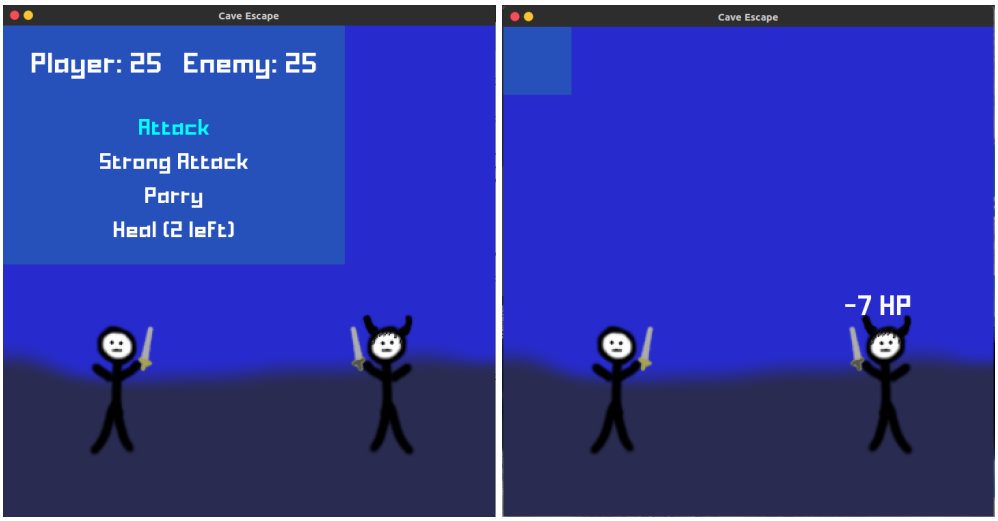

Figure 4.27: In-game screenshots from *Cave Escape*. The player character is situated on the left, while the opponent - an orc - is stationed on the right. The HP values of both player and AI are listed at the top of the screen. In the left image, the player is able to select their next move from the list of options. In the right image, the player has selected "Strong Attack" and the resulting damage is shown above the enemy's head.

An individual sprite was created for the player character and each of the five AI models, as shown in Figure 4.26. These sprites are overlaid on a bluish-purple background symbolizing the cave interior. The list of possible actions is also shown on-screen, positioned above the player character as in Figure 4.27. Next to the "Heal" option, the number of remaining healing turns is listed and updated after each turn. The HP of both players is also updated after every turn. Additionally, after players select an action, the resulting change in HP is shown above the head of the respective target. The right image in Figure 4.27 shows that the orc was hit by a strong attack made by the player, and took 7 points of damage.

Each non-player character has a respective AI model which it uses. The goblin uses the `randomAI()` function in Figure 4.19. The troll uses `aggressiveRandomAI()`, as seen in Figure 4.20. The orc uses the `fiftyPercentAI()` function from Figure 4.21. The elf is assigned to the `comparativeAI()` function seen in Figure 4.22, and the magician uses the `scalingDifficulty()` function from Figure 4.23. The baseline model in Figure 4.24 is assigned to the beserker, and the `parryCountering()` function in Figure 4.25 is assigned to the knight.

Figure 4.28: The set of sprites used when a character is parrying. Five of the characters are able to parry. Clockwise from top left: the player character, a goblin, an elf, a knight and a magician.

In addition to the default sprites shown in Figure 4.26, characters have a second sprite for when they are parrying. The Troll, Orc, and Beserker characters are unable to parry, so only five of these sprites were created. These variants are shown in Figure 4.28: the player character, the goblin, the elf, the knight, and the magician all have parry sprites.

At runtime, the game starts at a menu screen, where the five enemy characters are listed. When the HP of either the player or the AI falls to 0 or lower, the round ends. A final screen is shown to either celebrate the player's victory or lament their failure. The menu was made using code from a *Tetris* clone made with pygame created by Dimitris Strovolidis, released under the GNU General Public License, version 3 [21]. This code contains a `menu` class which allows both for interactive menus and easily-positioned text. The enemy characters are listed in this menu, and players are able to select which opponent they wish to fight. Both this title menu and the in-game action menus shown in Figure 4.27 use this `menu` class.

4.3 Playtesting and Survey

To test the effectiveness of these AI models, a group of participants were surveyed to play the various models. Each participant played five rounds against one of the models. The program collects data on wins and losses, as well as the sequence of actions taken by both the human player and the AI opponent. In addition to this game data, a short survey was given to each participant. Subjects were asked to answer the first three questions before playing the game. These questions asked participants to rate, on a scale of 1 to 5, their interests in video games, strategic board games, and RPGs, respectively. A score of 1 indicated little or no interest, while a score of 5 indicated a strong interest. After the three questions, participants were given a short description of the game, including the possible actions that players

could take in the game. After the participant completed the five rounds, they were asked three more questions. These questions asked them to rate, on a scale of 1 to 5, how fun they found the game, how understandable the controls were, and how aggressively or passively they played. 1 indicates a rating of not fun, that the controls were not understandable at all, and that the player used a very passive playstyle, respectively. A 5 rating indicated that the participant greatly enjoyed the game, that the controls were very understandable, and that their playstyle was very aggressive, respectively.

Most of the AI models were played by three participants, except the Goblin - which was tested by four participants - and the Beserker - which was tested by two participants. The responses given by the survey participants are recorded in Table 4.1. On average, survey participants rated themselves as having 3.5 out of 5 for interest in video games, 3.75 out of 5 for interest in strategic board games, and 3.5 out of 5 for interest in RPGs. In the post-game survey, participants rated the game 3.8 out of 5 for fun, and rated the controls a 4.6 out of 5 for ease of comprehension. For their own playstyle, players on average rated themselves 3.75 out of 5, indicating that the average playstyle was a fairly even mix of offensive and defensive choices, with a slight edge toward offensive.

Test #	AI Model	Q1	Q2	Q3	Q4	Q5	Q6
1	Goblin	3.5	5	4	3	5	5
2	Troll	5	5	5	4	5	3
3	Orc	4	4	5	5	5	3
4	Elf	4	5	4	4	5	1
5	Magician	5	3	4	5	5	4
6	Goblin	5	5	5	3	5	4
7	Troll	4	4	5	3	5	5
8	Orc	5	5	5	4	5	4
9	Elf	2	3	1	4	5	4
10	Magician	2	2	1	2	4	3
11	Goblin	4	5	4	5	3	5
12	Troll	1	2	1	2	3	3
13	Orc	3	3	5	5	5	3
14	Elf	3	4	2	3	5	5
15	Magician	4	3	3	4	5	4
16	Goblin	2	2	2	5	5	4
17	Beserker	5	3	5	4	5	3
18	Knight	5	4	4	5	5	4
19	Beserker	5	4	5	3	4	2
20	Knight	4	3	3	4	5	4
21	Knight	5	3	5	5	5	4

Table 4.1: Results of the survey given to all playtesters of *Cave Escape*. For each question, participants rated themselves on a scale of 1 to 5. The first three questions were asked before participants played the game. Q1 asks participants their interest in video games; Q2 asks their interest in strategic board games; Q3 asks their interest in RPGs. After participants played 5 rounds of the game, they answered the last three questions. Q4 asks participants their enjoyment of the game; Q5 asks their understanding of the controls; Q6 asks participants to rate the aggressiveness or defensiveness of their playstyle.

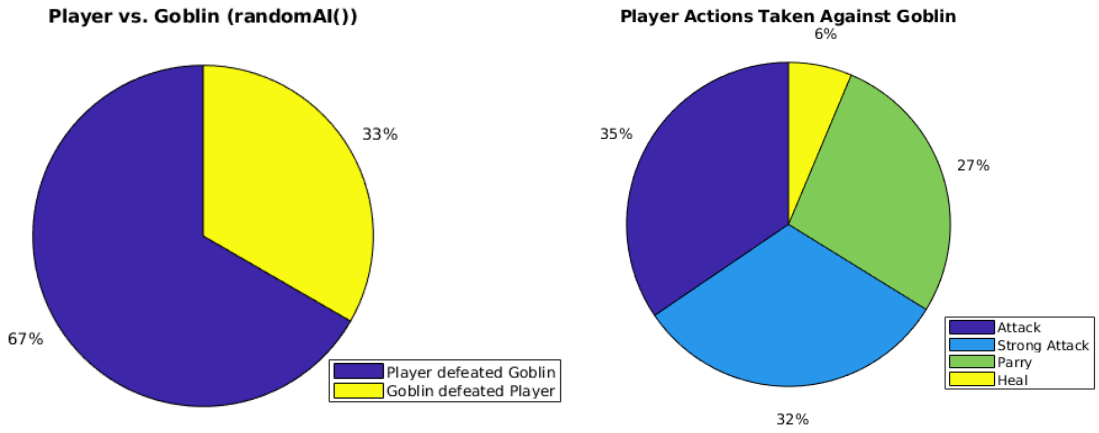

Figure 4.29: On left, the win/loss percentages for the randomAI() function, assigned to the Goblin character. On right, the percentage of actions players used against the Goblin.

After 20 individual games against the Goblin AI - using the randomAI function - the win percentage for the survey participants are as shown in the left chart in Figure 4.29: a 67% win percentage for the human player, and a 33% win percentage for the Goblin. Since this AI model uses a random number generator to determine its actions, this win percentage suggests that unpredictability is more difficult for human players to defeat. Furthermore, we can see that the expanded version of the game, with 25 HP instead of 10 and two more abilities, has more outcomes where player 2 can win.

Participants who played this AI model rated their enjoyment on average as 4 out of 5. These same participants rated their playstyle as 4.5 out of 5, indicating that they played very aggressively. However, as shown in the right chart of Figure 4.29, players actually had a fairly even split between normal attacks, strong attacks, and parries. Healing, since it is limited to twice per round, is necessarily less common. While as a whole players did opt to use offensive moves instead of defensive moves, their aggression is somewhat overstated.

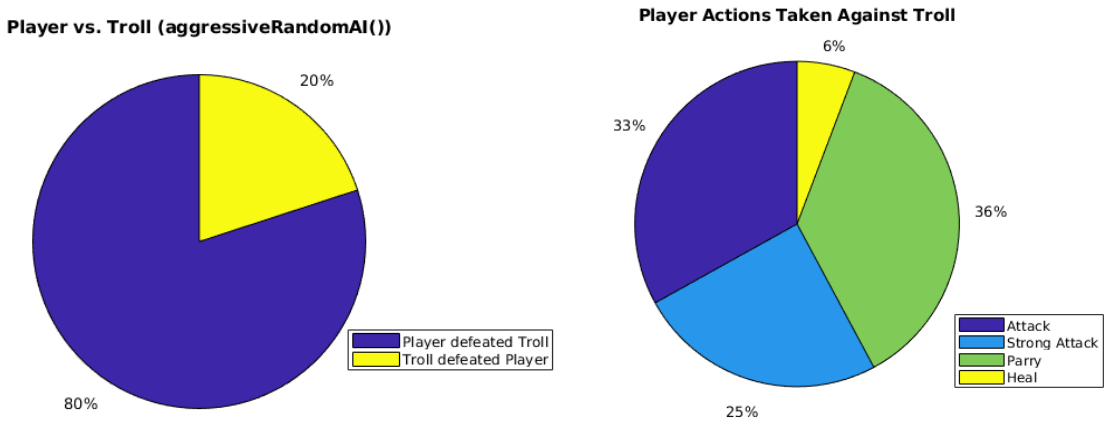

Figure 4.30: On left, the win/loss percentages for the aggressiveRandomAI() function, assigned to the Troll character. On right, the percentage of actions players used against the Troll.

The Troll AI fared much worse than the Goblin, as seen in the left chart of Figure 4.30. The aggressive AI was only able to win 20% of its games. In comparison to the Goblin, the Troll is less likely to use strong attacks; the Goblin can use a strong attack whenever, but the Troll must first expend both of their healing turns before it can use a strong attack. Additionally, the Goblin can parry while the Troll cannot. Thus, at any particular turn of the game, the Troll has a more limited move set than the Goblin.

Participants who played this AI model rated their enjoyment on average as 3 out of 5, the lowest of all the models. On average, they rated their playstyle as 2.75 out of 5, indicating that they leaned towards defensive playstyles. In comparison to the actions taken against the Goblin in the right chart of Figure 4.29, this self-identification is correct: players who fought the Troll tended to parry more than players who fought the Goblin, as shown on the right chart of Figure 4.30. Players who fought the Goblin parried 27% of the time, while players who fought the Troll parried 36% of the time.

Since the Troll primarily uses normal attacks, this increase in parrying is to be expected. With such an increase, we can infer that players were able to guess at

the decisions made by the Troll, and expected the Troll to use normal attacks. This is supported by the win/loss percentages as well, as players were more successful fighting the Troll than the Goblin.

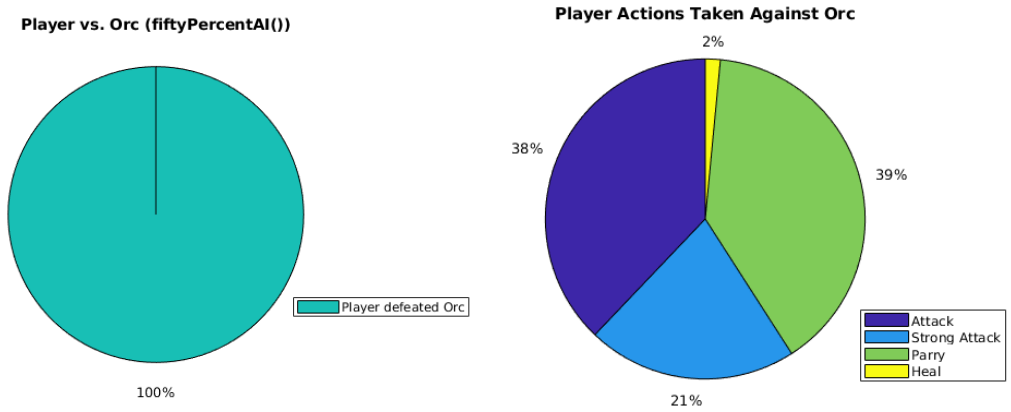

Figure 4.31: On left, the win/loss percentages for the fiftyPercentAI() function, assigned to the Orc character. On right, the percentage of actions players used against the Orc.

The Orc AI performed the worst of all, as is evident from the left chart of Figure 4.31. Not a single person lost to the Orc after 15 trials. The Orc also has the most limited move set of these first three models, only able to heal and use normal attacks. Seeing as the Orc has the same available moves as the players in the simple simulation in Figure 4.4, it is unsurprising that the Orc has trouble winning games.

Participants who played this AI model rated their enjoyment on average as 4.6 out of 5. They rated their playstyle as 3.3 out of 5, indicating that they leaned towards aggressive play, but were slightly more defensive than players who fought the Troll. As shown in the right chart of Figure 4.31, players who fought the Orc healed less often than players who fought the Goblin or Troll, but parried more than those same players. This data suggests that players did not feel as threatened by the Orc as they did the Troll or Goblin. As with the Troll, the Orc relies on normal attacks. Furthermore, unlike the Troll, the Orc does use any strong attacks which would counteract a player's parry. Thus, players can once again guess at the

decisions made by the Orc. Given that the Orc did not win a single battle, these players appear to be correct.

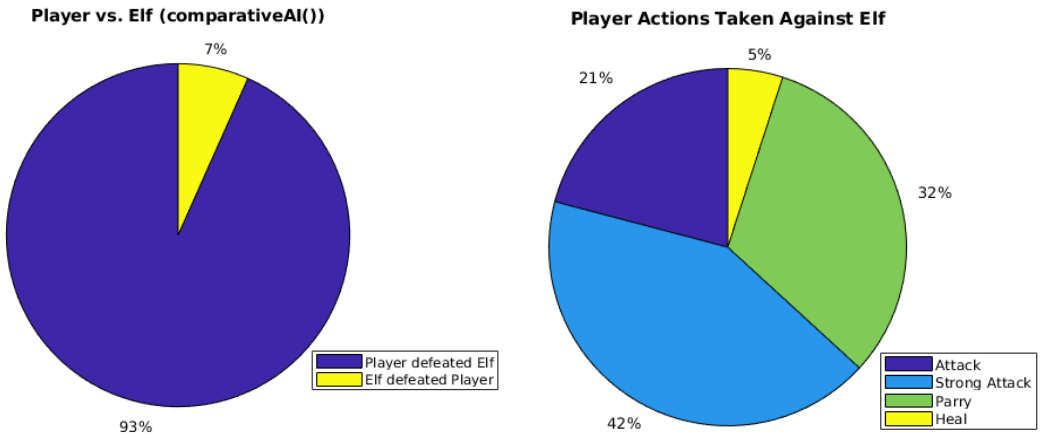

Figure 4.32: On left, the win/loss percentages for the comparativeAI() function, assigned to the Elf character. On right, the percentage of actions players used against the Elf.

The left chart of Figure 4.32 shows that the comparativeAI() function was only slightly successful. Human players won 93% of the games while the AI won 7% of the games. Since this model uses parries frequently, a player is able to safely heal themselves without immediately losing that HP on the Elf's next attack. Furthermore, a successful parry deals 3 points of damage while a successful strong attack deals 7 points of damage. If a player chooses to use a normal attack twice followed by a strong attack, they will lose 6 HP from the Elf's parries but will still have a net gain of 1 HP. Thus, a player can take a few hits from parries but still come out ahead with strong attacks. This difference gives players the opportunity to learn that strong attacks cannot be parried.

As seen in the right chart of Figure 4.32, players who fought the Elf used more strong attacks than against any of the AI opponents. Since the Elf tries to parry after it runs out of healing turns, the number of strong attacks predictably increases as players try to break through their opponent's parry. While not as low as against the Orc, players who fought the Elf chose to heal slightly less than against the Goblin

and Troll. Since the Elf is on the defense for much of each round, players are likely basing their decisions on ways to break through the Elf's parry and not on keeping their character alive. On average, players who fought this AI model rated their enjoyment of the game 3.5 out of 5. This rating is lower than the ratings for both the Goblin and Orc models, but higher than the rating for the Troll model. While the decisions made by the Elf are more complicated than the Goblin, the Elf's reliance on parrying impacted the enjoyment of the participants; participants seem to have more enjoyment playing AI models which were more active. Players rated their own strategy as a 3 out of 5 in terms of offensive and defensive. In comparison to the data, players actually used attacks (both normal and strong) 63% of the time, and used defensive moves (healing and parrying) 37% of the time. It is possible that, given that the Elf chooses to parry on most of its actions, players view this battle as more guarded than others. The Elf's defensive stance may give players the impression of a duel between two skilled opponents, with each looking to exploit an opening; this is in contrast with the Orc and Troll battles, where players alternate with attacks until one falls. Thus, even though the data tells a different story, players felt that they were more cautious when battling the Elf than against the Orc and Troll.

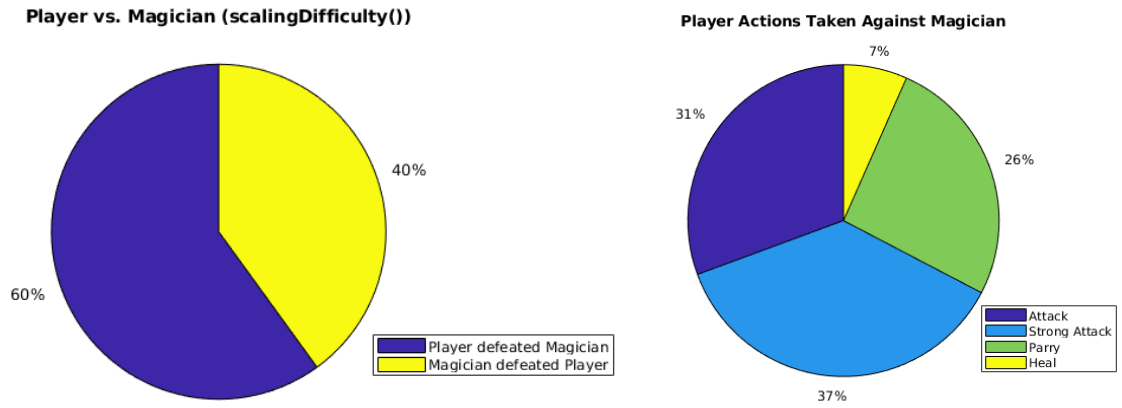

Figure 4.33: On left, the win/loss percentages for the scalingDifficulty() function, assigned to the Magician character. On right, the percentage of actions players used against the Magician.

The Magician character has one of the best overall records, as seen in the left chart of Figure 4.33: a win rate of 60% for the player and 40% for the Magician. Like the Goblin, the Magician is able to perform all four of the available actions, which the other three AI models could not do. Given the similar win/loss rates for the Goblin and Magician, it may be the case that, by limiting the available actions for the other AI models, they were handicapped in a way that the Goblin and Magician were not.

Players who fought the magician rated their enjoyment on average as 3.6 out of 5, and rated their playstyle as 3.6 out of 5 as well. This playstyle value indicates players chose offensive moves slightly more often than defensive moves. As with the Elf, the Magician's choice to parry at low health levels forces players to use strong attacks to win the game. Since the Magician parries less often than the Elf, players used fewer Strong Attacks against the Magician than they did against the Elf. Furthermore, players parried less often against the Magician than against the Elf. This data supports the self-identification of the participants' playstyle. Since the threshold for when the Magician starts parrying is lower than the same threshold for the Elf, players can afford to be more aggressive while fighting the Magician.

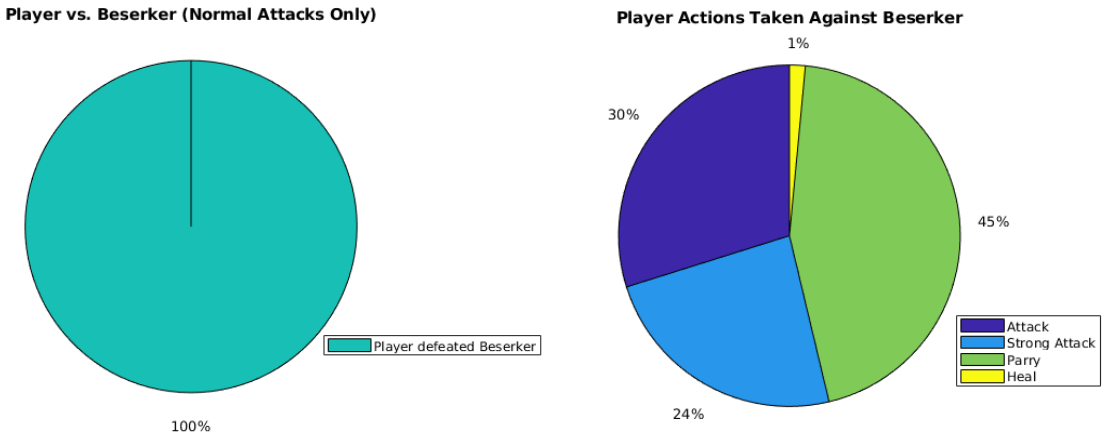

Figure 4.34: On left, the win/loss percentages for the baseline AI model, which only uses normal attacks. This AI model is designated as the Beserker. On right, the percentage of actions players used against the Beserker.

The Beserker, like the Orc, did not win a single battle. As a baseline model, the Beserker only used normal attacks. Compared to other AI models, it is most similar to the Orc, but without the healing turns taken when the model's HP drops below 50%. As shown in the right chart of Figure 4.34, players who fought this AI model chose to heal less often and parried more often against the Beserker than against any other AI model.

Players who fought the Beserker rated their enjoyment on average as 3.5 out of 5. These participants rated their playstyle as 2.5 out of 5, indicating a mostly defensive playstyle. This was the most defensive rating that players gave for any of the AI models. Given that these players chose to parry on 45% of their turns, the data supports the participants' ranking.

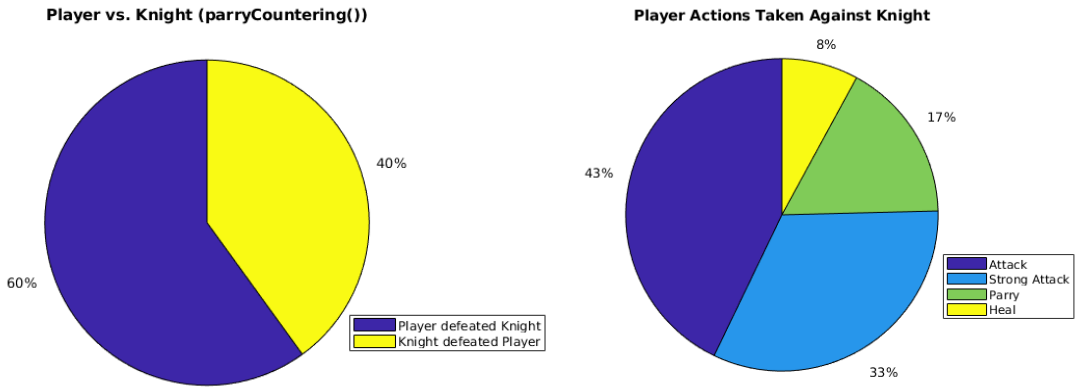

Figure 4.35: On left, the win/loss percentages for the `parryCountering()` function, assigned to the Knight character. On right, the percentage of actions players used against the Knight.

The Knight AI model performed comparatively to the Magician AI. As shown on the left chart of Figure 4.35, participants were able to defeat the Knight in 60% of the tests while the Knight defeated players in 40% of the tests.

Participants ranked their enjoyment on average as 4.6 out of 5; they ranked their playstyle as 4 out of 5, indicating mostly aggressive play. The only major difference between the Knight and the Magician models is the Knight's choice to use Strong Attacks any time that the player parries. As shown in the right chart of Figure 4.35, participants learned quickly not to attempt to parry; at 17%, players parried less often against the Knight than against any other AI model. Additionally, players healed more often while fighting the Knight than against any other AI model. In terms of offense, players used normal attacks more often than against any other AI model as well. In total, the data shows that players relied on the certainty of a normal attack, rather than deal with the possibility of a miss with a strong attack. This data, as well as the high ranking participants gave their aggression, suggests that players felt threatened by the Knight model.

While the player can choose their actions to play strategically, some of the AI models can be easily beaten if the player chooses to use normal attack every turn. In casual observation, the Troll, Orc, and Beserker can be reliably beaten by only

attacking. The Goblin can occasionally be defeated with only normal attacks, but will occasionally defeat the player. Like the early version of the game explored in Figure 4.4, the human player has a tremendous advantage by taking the first turn. However, the addition of parries and strong attacks allows for second player in this version of the game to win, while the second player in the first version could only win if Player 1 chose to heal at specific moments. If the human player chooses to only attack, the Magician, Elf, and Knight cannot be defeated. Since they stop parrying if their HP is above the player's HP, these AI models are able to stay ahead.

AI Model	Average Player HP
Goblin	6.350
Troll	7.067
Orc	11.867
Elf	19.467
Magician	7.000
Beserker	11.400
Knight	6.600

Table 4.2: The average HP value of the human player at the end of the game, separated by AI model.

Since some of the AI models had identical win/loss percentages to other models, we next examine the average HP of the players who fought each AI model. As shown in Table 4.2, players who fought the Elf suffered the least amount of damage on average at 19.467 hit points. Players who fought the Orc and Beserker models both had an average of around 11 HP at the end of the game. Although the Elf won more battles than either the Orc and Beserker, the Elf model dealt less damage overall. Thus, the players who fought the Elf and won did so without suffering much damage at all. Comparatively, the Orc and Beserker models were able to deal damage, but were defeated either with the player's strong attacks or parries.

Of the models, players who fought the Goblin had the lowest average HP. As a

random decision-maker, the Goblin does not have a strategy which the player can exploit. Of the non-random AI models, the Knight dealt the more damage to the player than the Magician: 6.600 compared to 7.000. Since these two models had equal win/loss percentages, this data indicates that battles against the Knight were more even than battles against the Magician.

4.4 Conclusion

We discuss the video game developed for this thesis, *Cave Escape*, and the AI models created as enemy players in the game. The AI models that were most successful in winning *Cave Escape* used some combination of all four available moves. The Goblin, Magician, and Knight models were all able to deal significant amounts of damage to the human player, and these AI models won in approximately 60 to 70% of their games. Of the four possible actions available to both human and computer player, healing was not a viable option. Human players could easily defeat the Beserker, Troll, and Orc characters by using only normal attacks or only parries.

CHAPTER 5

Future Work

We have seen with a limited survey sample that video game AI models are more challenging for players when they have larger sets of possible actions within the game. However, it would be useful to create a full extensive-tree model of the updated game, like the extensive tree created for the simpler version of the game. With such a tree, deeper analysis can be done on the effectiveness of each AI model. When the outcomes of the game are known, a more complex AI model can be created to specifically seek their winning outcomes. Additionally, this tree could help balance the game. If one player can reliably win using the same strategy each game, then certain parts of the game can be tweaked to mitigate such outcomes; for instance, the amount of damage dealt by a particular action could be decreased, or the probability of missing a strong attack could be increased. Given the random chance of missing strong attacks, there likely is not a simple solution to be found by backwards induction, but using the percentage of missed attacks in a mixed strategy could help to determine the optimal action.

Alternative machine learning models can be used to implement the AI. With the complete version of the game, an artificial neural network can be made to simulate the game and learn to play optimally. In some ways, this would be more efficient than attempting to solve the entire game in an extensive-form tree, but the neural network would not show why its decisions were optimal.

The balance of healing turns can also be adjusted for better results. Originally,

the values of 5 points of damage lost on an attack and 4 points of damage gained from healing were chosen to prevent infinitely long games in Figure 4.3. However, since later versions of the game limited both players to two healing turns per game, this issue does not occur. If *Cave Escape* changed the point value for healing turns, and the values of a normal attack and healing were tweaked accordingly, then it is likely that healing would become a much more viable strategy, as opposed to the current version where healing tends to be a detriment.

References

1. Necron (boss). http://finalfantasy.wikia.com/wiki/Necron_(boss). Accessed: 2018-11-29.

2. Carlos Alos-Ferrer and Klaus Ritzberger. Trees and extensive forms. *Journal of Economic Theory*, 143(1):216–250, 2008.

3. Ape and HAL Laboratory. Earthbound. Super Nintendo Entertainment System, 1994.

4. Atari. Asteroids. Arcade, 1979.

5. George J. Mailath and Larry Samuelson. *Repeated Games and Reputations*. Oxford University, 2006.

6. Tom Mitchell. *Machine Learning*. McGraw-Hill, 1997.

7. Monolith. Xenoblade Chronicles. Wii, 2010.

8. Namco. Galaga. Arcade, 1981.

9. Martin J. Osborne and Ariel Rubinstein. *A Course in Game Theory*. MIT Press, 1994.

10. Steve Rabin, editor. *Introduction to Game Development*. Charles River Media, Boston, MA, 2nd edition, 2010.

11. Thomas H Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

12. Jesse Schell. *The Art of Game Design*. CRC Press, Pittsburg, PA, 2015.

13. Brian Schwab. *AI Game Engine Programming*. Charles River Media, Hingham, MA, 2004.

14. Toby Segaran. *Programming Collective Intelligence*. O'Reilly, Sebastapol, CA, 2007.

15. Lloyd Shapley. Stochastic games. *Proceedings of the National Academy of Sciences of the United States of America*, 39:1095–1100, 1953.

16. Pete Shinners. Pygame documentation. `https://www.pygame.org/docs/`. Accessed: 2018-11-5.

17. Yoav Shoham and Kevin Leyton-Brown. *Multiagent Systems*. Cambridge University, New York, 2009.

18. Jouni Smed and Harri Hakonen. *Algorithms and Networking for Computer Games*. John Wiley & Sons, West Sussex, England, 2006.

19. Eilon Solan and Nicolas Vieille. Stochastic games. *Proceedings of the National Academy of Sciences of the United States of America*, 112:13743–13746, 2015.

20. Squaresoft. Final Fantasy IX. PlayStation, 2000.

21. Dimitris Strovolidis. Tetris. `https://gitlab.com/dimitrisstr/tetris`, 2018.

22. Taito. Space Invaders. Arcade, 1978.

23. Thomas J. Webster. *Analyzing Strategic Behavior in Business and Economics: A Game Theory Primer*. Lexington Books, 2014.

24. Lofti Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.