DYNAMIC DIFFICULTY ADJUSTMENT USING BEHAVIOR TREES

KENNETH SEJRSGAARD-JACOBSEN, TORKIL OLSEN AND LONG HUY PHAN

Master Thesis

Department of Computer Science
Aalborg University

June 2011

AALBORG UNIVERSITY
STUDENTS STUDYREPORT

Title: Dynamic Difficulty Adjustment Using Behavior Trees
Theme: Artificial Intelligence in Video Games
Project period: DAT6, Spring Semester 2011
Project group: f11d623a
Group members:

Kenneth Sejrsgaard-Jacobsen

Torkil Olsen

Long Huy Phan

Supervisor: Yifeng Zeng

Copies: 6
Number of pages: 58
Appendices: 3
Completion date: 9th of June 2011

Abstract:

We investigate the feasibility of adjusting the difficulty in a 2D fighting game using behavior trees. We present an overview of dynamic difficulty adjustment properties and define a fitness function as an appropriate measurement of difficulty based on a specific scenario. The purpose is to ensure a balanced game between a player and an agent by minimizing the difference between the player's skill and the difficulty of the agent. We propose two different approaches to adjust the difficulty of an agent, using behavior trees, based on the agent's fitness. The first method introduces a mechanism to switch between multiple predefined behavior trees to create variations in the difficulty. The second method utilizes a dynamic behavior tree where probabilities inherent in the tree structure are adjusted to change the behavior. Tests conducted show promising results for the second method.

CONTENTS

1 INTRODUCTION 1
1.1 Related Work 2
1.2 Problem Statement 4
2 M.U.G.E.N 7
2.1 2D Fighting Game - xnaMugen 7
2.2 Game Scenario 9
2.3 Difficulty Parameters 9
2.3.1 Range 10
2.3.2 Delay 10
3 BEHAVIOR TREES 11
3.1 Non-leaf Nodes 11
3.2 Leaf Nodes 12
3.3 Example 13
4 DYNAMIC DIFFICULTY ADJUSTMENT 15
4.1 Definitions 16
4.2 Fitness 16
4.2.1 Fitness Function 17
5 BEHAVIOR DESIGN 19
5.1 Static Behaviors 19
5.1.1 Behaviors in $\mathcal{B}$ 20
5.1.2 Behaviors in $\mathcal{C}$ 23
6 DYNAMIC DIFFICULTY ADJUSTMENT USING MULTIPLE BEHAVIOR TREES 25
6.1 Approach 26
6.2 Algorithm 27
6.3 Results 28
6.3.1 Setup 28
6.3.2 DDA1 Versus $\mathcal{B}$ Agents 29
6.3.3 DDA1 Versus $\mathcal{C}$ Agents 32
6.3.4 Average End Fitness of DDA1 32
7 DYNAMIC DIFFICULTY ADJUSTMENT USING DYNAMIC BEHAVIOR TREES 35
7.1 Linear Programming 35
7.2 Quantification of Agent Outcome 36
7.2.1 Utility 36
7.2.2 Expected Utility 36
7.3 Approach 37
7.3.1 Example 39
7.4 Results 40
7.4.1 Setup 41
7.4.2 DDA2 Versus B Agents 41
7.4.3 DDA2 Versus C Agents 43
7.4.4 Average End Fitness of DDA2 44
7.4.5 Summary 45
8 FUTURE WORK 47

9 CONCLUSION 49
A APPENDIX A 51
B APPENDIX B 53
C APPENDIX C 55

BIBLIOGRAPHY 57

LIST OF FIGURES

Figure 1.1 Flow model [1] . . . 1
Figure 2.1 Remaining time as well as health and power bars can be seen in the figure. . . . 8
Figure 2.2 A powerful punch combo (a) and kick combo (b). . . . 8
Figure 2.3 Punch attack animation. . . . 9
Figure 3.1 Root . . . 11
Figure 3.2 Selector . . . 11
Figure 3.3 Probability Selector . . . 12
Figure 3.4 Sequence . . . 12
Figure 3.5 Decorator . . . 12
Figure 3.6 Condition . . . 12
Figure 3.7 Action . . . 13
Figure 3.8 Link . . . 13
Figure 3.9 Behavior Tree (BT) structure. . . . 13
Figure 4.1 Flow channel in our xnaMugen scenario. . . . 15
Figure 4.2 The fitness $f(\rho)_t$ of an agent and $f(\rho')_t$ of a player. . . . 17
Figure 5.1 BT structure. . . . 19
Figure 6.1 Illustration of the intervals of three different BTs. . . . 26
Figure 6.2 Results from *DDA1* versus the BTs from $\mathcal{B}$. . . . 29
Figure 6.3 Health points through a match between *DDA1* and *B4*. . . . 30
Figure 6.4 A graph that shows when *DDA1* changes behavior against *B2* over the course of a game. The behavior intervals are the areas encapsulated between two lines. . . . 31
Figure 6.5 A graph that shows when *DDA1* changes behavior against *B4* over the course of a game. The behavior intervals are the areas encapsulated between two lines. . . . 31
Figure 6.6 Results of *DDA1* against the BTs from $\mathcal{C}$. . . . 32
Figure 7.1 Results for matches between *DDA2* and the $\mathcal{B}$ agents. . . . 41
Figure 7.2 Fitness values in a *DDA2* versus *B5* fight. . . . 42
Figure 7.3 Health points in a *DDA2* versus *B5* fight. . . . 42
Figure 7.4 Probability distribution of attacks in a *DDA2* versus *B5* fight. . . . 43
Figure 7.5 Results from *DDA2* versus the $\mathcal{C}$ agents. . . . 44
Figure C.1 Comparison of results for *DDA1* and *DDA2* against *B1* (a) and *B2* (b). 55
Figure C.2 Comparison of results for *DDA1* and *DDA2* against *B3* (a) and *B4* (b). 55
Figure C.3 Comparison of results for *DDA1* and *DDA2* against *B5* (a) and *Random* $\mathcal{B}$ (b). . . . 55
Figure C.4 Comparison of results for *DDA1* and *DDA2* against *C1* (a) and *C2* (b). 56
Figure C.5 Comparison of results for *DDA1* and *DDA2* against *C3* (a) and *C4* (b). 56

Figure C.6 Comparison of results for *DDA1* and *DDA2* against *C5* (a) and *Random* $\mathcal{B}$ (b). 56

LIST OF TABLES

Table 5.1 Win-loss ratio between *Weak Punch only agent* and *Strong Punch only agent*. 20
Table 5.2 Probability distribution in the attack selector node for each BT in $\mathcal{B}$. . 20
Table 5.3 B1 results. 21
Table 5.4 B2 results. 21
Table 5.5 B3 results. 22
Table 5.6 B4 results. 22
Table 5.7 B5 results. 23
Table 5.8 Probability distribution in the attack selector node for each BT in $\mathcal{C}$. . 23
Table 6.1 The results of the *DDA1* against the BTs from $\mathcal{B}$. 29
Table 6.2 The results of *DDA1* against the BTs from $\mathcal{C}$. 32
Table 6.3 Comparison of Average End Fitness (AEF) values for *DDA1* against the $\mathcal{B}$ and $\mathcal{C}$ agents. 33
Table 7.1 Results for matches between *DDA2* and the $\mathcal{B}$ agents. 41
Table 7.2 Results from *DDA2* versus the $\mathcal{C}$ agents. 44
Table 7.3 Comparison of AEF values for *DDA2* against the $\mathcal{B}$ and $\mathcal{C}$ agents. . . . 45
Table A.1 *C1* (a) and *C2* (b) results. 51
Table A.2 *C3* (a) and *C4* (b) results. 51
Table A.3 *C5* results. 51

ACRONYMS

DDA Dynamic Difficulty Adjustment

BT Behavior Tree

EF End Fitness

AEF Average End Fitness

LP Linear Programming

AFI Active Fitness Interval

AI Artificial Intelligence

HFSM Hierarchical Finite State Machine

RTS Real-Time Strategy

FPS First Person Shooter

1 INTRODUCTION

Video games are all about challenge and entertainment. Few people find video games entertaining if they do not pose a challenge. If they are too easy we find them boring and when they are too challenging we get frustrated. Most single player games provide a *static difficulty setting*, which determines the difficulty of a game. Usually the setting ranges from easy to normal to hard. The purpose of the difficulty setting is to accommodate both new and experienced players, such that they are all challenged in the game. The pros of a static difficulty setting is that it usually can be implemented as an adjustment of some variables, e.g., the health and strength of the opponents. In that way higher difficulty settings will make the opponents deal more damage and take more hits before going down.

When game difficulty is static it can lead to inconsistencies between the skill level of the player and the challenge of the game. To address this problem, Dynamic Difficulty Adjustment (DDA) was introduced. DDA is an alternative to static difficulty settings and is a concept which has gained more attention in recent years. The general idea with DDA is to adjust the game difficulty in-game according to the player's skill level, instead of relying on a preset static difficulty setting chosen by the player. The purpose is to present the player with a challenge, no matter what skill level he is at, and in that sense provide the player with a consistent entertaining experience.

This relates to the *flow model*, Figure 1.1, proposed by M. Csikszentmihalyi [6], where the goal is to obtain a balance between the skills and the challenge. It explains, how people are in a state of flow, when the challenge given to them corresponds with their personal level of competence. This easily maps to the challenges presented in a game and the skill level of the person playing the game. Looking at Figure 1.1 the preferred state is when the challenge matches the skills. This is also known as the *flow channel*. By constantly adjusting the challenge to the skills of the player he will never be left in a state of anxiety or boredom. This is what DDA aims to achieve. There are many ways to apply DDA to games such as, adjustment of behavior, changes of the game environment, etc.

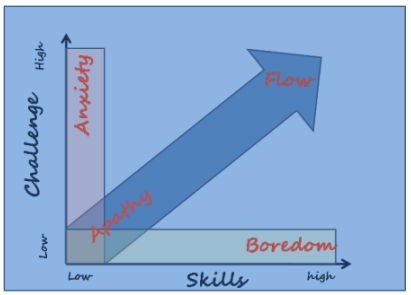


Figure 1.1: Flow model [1]

Several implementations of DDA exist in the industry. In Half-Life 2 [24], a First Person Shooter (FPS) game, more ammunition was dropped by the enemies if the player was running low on ammunition and if the player was on the verge of death, there would be a greater probability of finding big health kits. Subtle changes that were unobtrusive and invisible to the player. It is said that the best implementations of DDA are those that are unnoticed by the player [23].

Some of the less successful attempts in applying unobtrusive DDA can be seen in most racing games. Most people who have ever played a racing game have found themselves in a certain win situation with all the opponents being far behind, only to witness them all fly by, as soon as a mistake is made. This is known as the *Rubber band AI*, a poor implementation of DDA that is disliked among many gamers, as the player is well aware of the Artificial Intelligence (AI) cheating. The rubber band effect can be viewed as a rubber band connecting the player to his opponents. The farther ahead the player gets the faster the opponents will go; they will even go faster than the cars can actually go. This lets them catch up in the blink of an eye if the player crashes his car. If the player gets behind, the opponents will slow down to let the player catch up. The idea behind this is to make the player feel as if it is a close race. A very good idea, but often poorly executed.

Although DDA, when applied correctly, sounds very nice, it still has its problems. The more complex a game is, the harder it is to apply a suitable DDA to it. Furthermore, it is difficult to verify that the AI behaves as intended. With static difficulty the AI can not act in any unforeseen way, but by adding DDA the behavior of the AI can become unpredictable.

AI is quickly becoming a larger part of computer game development today. With graphics approaching photorealism, the next big improvement is the AI experience. However, there are several issues one must deal with when developing game AI. First off, games today are often very complex with respect to the size of the game world and the realism demanded of NPCs. Furthermore, AIs should provide equal entertainment for all players regardless of their individual skill level and experience with computer games.

1.1 RELATED WORK

To get a grasp on what is going on in the scientific community on the topic of DDA, we take a brief look at work done recently. We want to find a definition of difficulty and why it is important for games to be able to adjust to the players playing the game.

Aponte et al. [4] point out that automated play testing with synthetic players is a great cost and time efficient alternative to human player testing, but obviously not as good. Furthermore, they state that the progression of game difficulty relies on two sets of challenges: A set of basic challenges whose complexity can be controlled through a set of parameters, and an ordered set of challenges. They define a general difficulty evaluation function for games which calculates the probability for a player completing a challenge in a time limit given the players history in the game.

Andrade et al. [2, 3] approach the DDA problem in a 2D fighting game with Q-learning and a challenge function. A regular Q-learning agent would want to execute the action with best Q value in order to act as efficiently as possible, but that is not the goal in DDA. This is where the challenge function is introduced. The challenge function forces the agent to pick the second best or third best action and so on until the agent matches the human player's skill level. The challenge function is based on the difference between the players' health. The authors' assumption is that a balanced game is when the challenge function fluctuates around zero.

Another application of DDA has been made for the popular video game *Half Life* by R. Hunicke and V. Chapman [12]. Using a tool, Hamlet, they observe the damage the player takes and predicts whether intervention is necessary to reduce or increase the difficulty of the game. The goal of their approach is to make adjustments only when necessary as well as to make sure the adjustments are as seamless as possible.

Lankveld et al. [25] discuss how game balancing can be implemented supported by the theory of incongruity. Incongruity describes the difference between the contextual and internal complexity of a game. The contextual complexity in games consist of the game environment while the internal complexity describes the players tactical ability with respect to the game as well as his control of the game interface. By subtracting the mental complexity of the player from the contextual complexity of the game they calculate the incongruity which represents the difficulty of the game. An incongruity with a large negative value indicates a game which poses no challenge to the player whilst the opposite indicates that the game is too difficult. According to the authors the purpose is to balance the game s.t. the incongruity maintains a value close to zero ensuring a continuous challenge for the player leaving him in a state of flow.

Hagelbäck et al. [11] investigate whether people think it is more entertaining to win all the time or whether it is more entertaining to be challenged and play an even game against the AI. The game they use for their case study is *Open Real Time Strategy*. It was an investigation conducted under Dreamhack 2008, where 60 people participated. They had created a few static and dynamic agents. The dynamic agent adapted to the human player by making some of its units idle in several time frames. They did this using probabilistic models. When the human player was losing, the agent's units became idle for some time frames. The results they obtained were that the static agents were either too easy or too difficult and the dynamic agents were the most entertaining to play against.

Spronck et al. [20] investigate whether dynamic scripting can be used to create a balanced game between the human player and the game AI. Dynamic scripting is a technique that is able to automatically optimize game AI in game. They investigate three difficulty scaling methods; high fitness penalizing, weight clipping and top culling. The game they use as their testbed is a simulation of the *Baldur's Gate* game, simulating a two versus two scenario. The authors conclude that dynamic scripting is successful in creating an even game when top culling is utilized.

Verma et al. [26] propose an adaptive methodology using a genetic algorithm to create variations in a cellphone video game. The game they use for their scenario is an adaptation of the *Snake* game. After the human player finishes a game, he can decide to either play an easier or more difficult game depending on his skill level. This request will be sent to the server that runs the genetic algorithm. The genetic algorithm uses a fitness function to determine the skill level of the human player. It then responds by dispatching a game that is more suitable for the human player. The authors conclude that the methodology is successful with 90% of the participants agreeing that the game matched their desired difficulty.

Yannakakis et al. [17] generate neuro-evolved agents using Neuro-Evolution of Augmenting Topologies and real-time NEAT for a Real-Time Strategy (RTS) game. The goal is to evolve agents in real time that fit to the human player's skill level. A challenge rating of the game is defined s.t. a player's skill level can be quantified. The agents are then evolved using a fitness function which yields a high fitness if the difference in the challenge rating between the player and agent is minimal. The agents evolved will ideally have approximately the same challenge rating as the opponent player.

According to the related work there seems to be consensus on what a balanced game is, namely when the challenge of the game matches the skill of the player. This requires a measurement of both the game challenge and the player's skill. To be able to adjust the challenge of the game the variables affecting the challenge also need to be identified. What is left is to minimize the difference between the difficulty rating of the game and skill level of the player. Another important aspect of DDA is that the adjustments made in the game should be as seamless as possible to the player. Few of the results from aforementioned work have been cross verified using human test subjects as it is a costly and time consuming task. However, testing using synthetic agents provide useful and practical results sufficient as a starting point. We too find testing with synthetic agents adequate for this project.

Being a new concept there does not exist a lot of material on BTs and moreover it is difficult to find a formal definition of BTs. In *Getting Started with Decision Making and Control Systems* [19, sec.3.4], Alex J. Champandard describes the general concepts of a BT framework. Though the article's main focus is on the implementation of a BT framework, the author does make some arguments for the advantages of using BTs in place of Hierarchical Finite State Machines (HFSMs), mainly their simplicity and modularity. The article also briefly describes the basic building blocks of a BT referred to as *composite tasks* and explains how the decomposition of tasks is essential to the construction of a good BT. This article is one of the few literatures we have found on BTs and have been the source of inspiration for our own framework.

1.2 PROBLEM STATEMENT

We will investigate the usage of BTs to dynamically adjust the difficulty of a 2D fighting game. We have chosen BTs because it is a method, which has recently gained attention in the game industry. Furthermore, BTs provide an intuitive way of modeling the behavior of single actors, such as a character in a 2D fighting game.

We propose two different approaches for DDA in a simple scenario. Common for each approach is the definition of a balanced game and the circumstances under which adjustment of the difficulty is needed. As the human player plays the game, his performance will be measured and compared to the difficulty of the game. Based on this comparison, adjustments will be made to the agent, to fit the appropriate difficulty according to the player's skill.

The proposed methods require the consideration of the following issues:

- A proper measurement of the skill level of the human player needs to be defined. The metric should take into consideration measurable aspects of the human player's performance.

- A proper measurement of the difficulty of the game needs to be defined. The metric is based on the performance of the agent.

- The properties of the BT affecting the difficulty of the game need to be identified. Methods for adjusting the agent to balance the difficulty of the game and the player's skill level needs to be defined. By adjusting to the player's skill level the methods should ensure an even match up at all times.

To address these issues we specify a game scenario for testing. In this scenario we will identify the variables affecting the difficulty of the game and find a measurement of the agent's performance. Through further analysis of these elements, we will devise methods for dynamic adjustment of the agent's behavior based on the skill level of the player.

2

M.U.G.E.N

In this chapter we give a description of the game environment chosen for this project, as well as how we adjust the game environment to create a suitable game scenario. The game scenario will form the basis of the evaluation of the proposed methods.

M.U.G.E.N is a 2D fighting game engine designed by Elecbyte and originally released in 1999. It allows users to create and add new content, as well as modify existing game content, such as characters, stages, game screens, sound effects and more. Characters can be configured to use up to seven buttons and can employ regular moves, special moves, super moves, combos, projectiles, throws etc. All in all the M.U.G.E.N engine enables the creation of commercial-quality 2D fighting games. The engine comes with a 2D fighting game with the same name, M.U.G.E.N, with a single playable character called Kung Fu Man [7].

As the M.U.G.E.N source code is unavailable it puts some limitations on what can and cannot be done with the engine. In our case specifically, it puts some limitations on the control that we have over an AI in the game. Although, it is possible to develop AIs to use in the engine with scripting, it does not give us adequate control. We need a development environment where we have a more fine grained control over the engine. The AI needs to be controlled by a BT and furthermore, we need to be able to monitor and run diagnostics on the game.

To overcome these limitations we use *xnaMugen*, a clone of the M.U.G.E.N engine [10]. XnaMugen emulates the M.U.G.E.N engine using the .NET 2.0 and XNA 3.0 framework. The source code, written in C#, is available and licensed under the BSD license. The game implementation in xnaMugen currently only supports the versus game mode, where two human players take control of one character each and fight each other. In order to incorporate AI into the game, we simulate key presses to take control of one or more players instead of letting them be controlled by keyboard input.

2.1 2D FIGHTING GAME - XNAMUGEN

A 2D fighting game is a game, where two players are set to fight each other in a two dimensional arena. When looking at play style and game mechanics, xnaMugen is very similar to famous titles representing this game genre like, Team Ninja's Dead or Alive series [22], Midway's Mortal Kombat series [15] and Namco's Tekken series [16] to name a few. The objective of the game is to win a match, which consists of a best of tree rounds with a fixed time limit on rounds. The match can be won in two ways. One is by getting the opponents health points, represented as a yellow health bar in the top of the screen as seen on Figure 2.1, to zero. The other is by staying alive and retaining more health points than the opponent when the round timer hits zero. The match takes place in a stage of the players' choosing and prior to the match, the players each pick a character to use in the match. The stage is a two dimensional arena where the players can move freely horizontally,

and any movement on the vertical axis is achieved by a jump or crouch move. In xnaMugen the environment is an enclosed static arena with a fixed size. Some fighting games have dynamic stages, where the environment plays a role in the fight by having ledges or stairs that players can be kicked off to take additional damage.

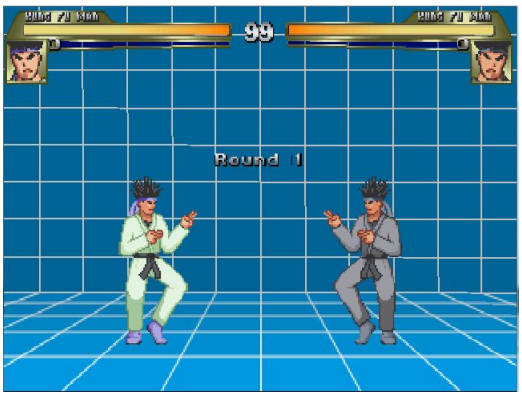


Figure 2.1: Remaining time as well as health and power bars can be seen in the figure.

Players control their characters with the directional buttons, up, down, left, right and have four buttons to perform two types of punches and kicks, the weak but fast attacks and the strong but slow attacks. Attacks can be performed while standing, crouching or jumping. By combining movement and attack commands in a specific order during a small time frame, players can perform powerful combos. Some combos require power to be executed. The power of the players is represented as a blue power bar beneath the health bar and the power increases when hitting or being hit and decreases when used for combos. Some combos get increased damage if power is available, but are not dependent on power in order to be executed. Figure 2.2 shows an example of two powerful attack combos.

(a) (b)

Figure 2.2: A powerful punch combo (a) and kick combo (b).

In order to block an attack, one can move backwards away from the opponent when he attacks, which will result in a blocking move. Doing this while the opponent is not attacking will make the character walk backwards. This move can be executed while standing, crouching or jumping.

2.2 GAME SCENARIO

We will not be using a full game scenario of xnaMugen. Games have a tendency of becoming very complex very fast and this is the reason for scaling it down. If there are too many dimensions and elements in the game, the game becomes increasingly difficult to study and run simulations on with viable results. First of all, we restrict a match to be one round with no time limit. Furthermore, the roster will be restricted to one character, Kung Fu Man. In similar games, there are usually several characters to choose from, all employing a different arsenal of special attacks and combos, but allowing more characters, only adds to the complexity of the game. In order to make the matches last longer, we double the total life of the agents to 2000 health points. By doing this, it is possible to make the game fluctuate in both players favor.

Besides the aforementioned alterations we restrict the game even further. As such, the agent will only utilize a subset of the actions available in the full game. Agents will only be able to move on the horizontal axis, as they are not allowed to perform jump or duck actions. The number of attack actions have been reduced down to two attacks. These attack actions are weak punch and strong punch. The two attacks are identical except for the damage they deal. The weak punch deals 25 damage and the strong deals 50 damage. The attack is illustrated in Figure 2.3.

Figure 2.3: Punch attack animation.

The reason for putting these restrictions on the game is merely to reduce the complexity of the game. The restrictions on available game actions for agents will result in matches with a low degree of variety, but the game will be more manageable and easier to analyze. We feel that this game scenario provides a sufficient testbed for the purpose of showing the feasibility of the approaches presented. The goal is not to design human-like agents with a huge variety in behavior, but to apply DDA with the goal of creating a balanced game. Variety is of course required for great game play, but it is not essential for this project.

2.3 DIFFICULTY PARAMETERS

We have identified some parameters that have a significant influence on the difficulty of the agents. The parameters identified are range and delay. By running some simulations with agents utilizing the same behavior, but with slightly different values for these parameters,

we observed a remarkable difference. Agents with a delay approaching zero and a range approaching the maximum range of specific attack actions showed increasing performance. It is no surprise to see these parameters influence the performance of the agents, as exactly the same parameters influence the performance of human players.

2.3.1 *Range*

Range is used all across the BTs in condition nodes, e.g., a condition node prior to an attack action, to check whether the opponent is close enough to be hit by the attack. The range parameter can be seen, as how well a player perceives the state of the game world and the opponent. A player with good perception will know better, when to perform an attack depending on the distance to the opponent. If the range parameter is set to the maximum range of the specific attacks the agent can perform, then it will be able to perform the attacks the moment his opponent gets inside the range.

2.3.2 *Delay*

The delay specifies the delay after the agent has finished an attack sequence of actions, e.g., a combo, before executing a new action. Without any delay the agents would act unrealistically fast, so in order to add some sense of realism to the agents, the delay is required. The delay parameter maps very nicely to the reaction and response time of a human player. Obviously a player with a short reaction time will have an advantage over a slower player. A slow player may know the most powerful moves and combos, but if challenged by a fast player only performing basic attacks, he would have a hard time getting through his well timed defensive and offensive moves.

These parameters are very game specific and do not influence the generality of the proposed methods. We want to investigate how BTs can be utilized to adjust the difficulty. Thus we abstract from the specific parameters by choosing two attacks with the same characteristics, using the same range and delay parameter for both attacks.

3

BEHAVIOR TREES

The syntax we have chosen for BTs is based on the notation proposed by Alex J. Champandard [19]. In general BTs can be specified as a hierarchical structure of tasks. As such, a BT can be seen as a structure that stores a plan for the agent's behavior. Opposed to planners a BT is static and only stores the plan which can be traversed by the agent; it does not perform any kind of planning. Some work have been done on expanding BTs with a new node construct that queries a library of actions depending on the world state [9]. Although a clever addition, we find the basic constructs of BTs sufficient for this project.

A BT starts with a single *root* node specifying the beginning of the tree. The non-leaf nodes can be viewed as a composition of tasks. A task can be a composite of additional non-leaf nodes as well as leaf nodes. When traversing down the tree, the tasks get more specific and finally end up in specific actions that are executed in the game environment. The execution of a BT is done depth-first, starting from the root node. Usually, all non-leaf nodes will execute their children from left to right, but there are exceptions depending on the node type. When a node has finished executing, it returns a status which can either be success or failure. The circumstances under which success or failure is returned, depends on the node type.

3.1 NON-LEAF NODES

The non-leaf nodes can each have any finite number of children. There are two non-leaf nodes that act as complements of each other. These are the *selectors* and *sequences*. Additionally, we have the *decorator*.

Root

The root node, Figure 3.1, is not an actual node in the tree. It serves as the entry point of the tree and can only have one child, which can either be a non-leaf node or a leaf node. A BT can have only one root node in its structure.

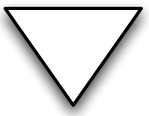


Figure 3.1: Root

Selector

The selector node, Figure 3.2, will sequentially try to execute its child nodes from left to right, until it receives a successful response. When a successful response is received, it responds to its parent with a success. If a child node responds with a failure, the selector node executes the next child node in line. If all child nodes respond with failure, the selector node itself responds with a failure. The selector can be viewed as an OR construct where only one child needs to succeed.

Figure 3.2: Selector

Probability Selector

A probability selector, Figure 3.3, is a selector node with a probability distribution over its children. The probability indicates how likely a child is to be chosen during execution. If the child node chosen responds with a failure, the selector will normalize the probability distribution over the remaining children and choose a new child to be executed. It responds to its parent in the same way as a normal selector node.

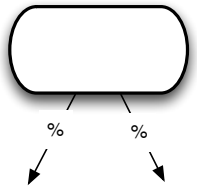


Figure 3.3: Probability Selector

Sequence

The sequence node, Figure 3.4, will execute each of its child nodes in sequence from left to right. If every child node responds with a success the sequence node itself will respond with a success to its parent. If somewhere during the sequence of execution, a child node responds with a failure, the sequence node will respond with a failure. The sequence node can be viewed as an AND construct where all children must succeed.


Figure 3.4: Sequence

Decorator

The decorator node can be added to the BT to provide additional functionality to behaviors. In contrast to other composite nodes, decorators can only have one child node. Decorators are commonly used for functionality, such as filtering using counters or timers. A decorator can implement many different features and it is recommended to do so, if it keeps the other composite nodes simple. Figure 3.5 illustrates a decorator node.

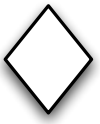


Figure 3.5: Decorator

3.2 LEAF NODES

The leaf nodes are called *condition*, *action* and *link* nodes. The leaf nodes except for the link node in a BT specify an observation or interaction with the game environment. These leaf nodes can be viewed as primitive tasks as this is where the BT interacts with the game environment.

Condition

Conditions, Figure 3.6, are nodes that will observe the state of the game environment and respond with either a success or a failure, based on the observation. This could be a condition check, value comparison, etc.

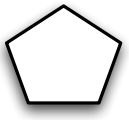


Figure 3.6: Condition

Action

Action nodes, Figure 3.7, are used to interact with the game environment. Through actions we can control various aspects of a game, such as character movement, interaction with objects, etc. When actions are performed successfully, the action node will respond with a success. If the game engine fails to perform the action, the action node will respond with a failure.

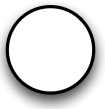


Figure 3.7: Action

Link

A link node, Figure 3.8, holds a link to the root of another BT. When a link node is executed it will execute the linked BT and wait for a response. If the linked BT is successfully executed, it will respond with success, otherwise failure. This introduces modularity and reusability of behaviors.

Figure 3.8: Link

3.3 EXAMPLE

To give the reader a better understanding of how BTs work, we will go through an example of a BT in detail. The BT shown in Figure 3.9 is the tree structure that we are going to use throughout the report. We will go through different scenarios, beginning with a successful traversal of the BT.

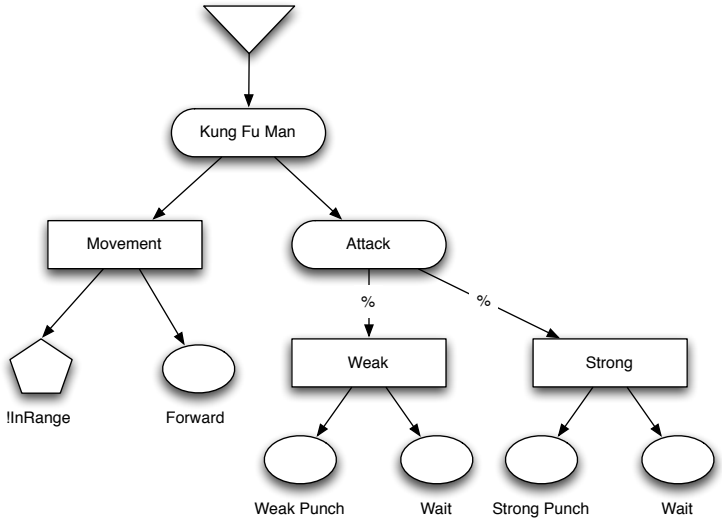


Figure 3.9: BT structure.

It should be noted that selectors and sequences usually do not have labels, however to make it easier to grasp the concept of BTs, we decided to include labels for those constructs. Furthermore, it will make it easier to discuss the BT. When traversing a BT, we always start from the root. We continue by executing its child, in this case it is the selector *Kung Fu Man*. When the *Kung Fu Man* selector has been executed, we continue to execute its first child, the sequence *Movement*. *Movement* will execute its children from left to right, beginning with

the condition *!InRange*. This condition checks whether we are '*not*' in range of the opponent that is, outside of range. If that is true, we execute the *Forward* action. Assuming that this traversal is a successful one, both children will return with a success status to their parent. As both of the *Movement* sequence's children succeeded, *Movement* will succeed as well. *Kung Fu Man* will succeed and the traversal terminates with a success status.

Let us now assume *Movement* failed. *Kung Fu Man* will then continue by executing its next child, the *Attack* probability selector. Assume the *Attack* probability selector has a probability distribution of 0.8/0.2. We will have the highest probability of executing the *Weak* sequence. *Weak* will start executing the action *Weak Punch* followed by the *Wait* action. Now *Wait* fails which causes it to return a failure status to its parent *Weak*. As one of *Weak's* children failed, *Weak* will also fail. This will cause *Attack* to normalize the probability distribution over its remaining children, resulting in a certain execution of *Strong*. If we assume that the *Wait* child of *Strong* returns with a failure as well that will cause *Strong* to fail followed by *Attack* and *Kung Fu Man*. Finally the traversal of the tree fails and terminates with a failure status.

4 DYNAMIC DIFFICULTY ADJUSTMENT

DDA can be achieved in many ways and is highly dependent on the scenario and approach. In this chapter we discuss the basics of DDA and the theory we apply to our scenario. Following this general description, we will explain our definition of a fitness function in relation to our game scenario and which criteria should be met for the DDA to be successful.

The overall goal of our DDA approach is to make the game balanced, during the entirety of the fight. Furthermore, a series of matches should result in an approximately even amount of wins for each agent. We consider a game balanced, if the challenge of the game match the player's skill in relation to the flow model [6]. We make the assumption that our scenario is balanced when both players have roughly the same amount of health points, illustrated as the flow channel in Figure 4.1. A game where the agent has considerably more health points than the player, may cause anxiety for the player, while the opposite may result in a feeling of boredom. This definition is very specific to our scenario, but we believe it is adequate for the purpose of this project.

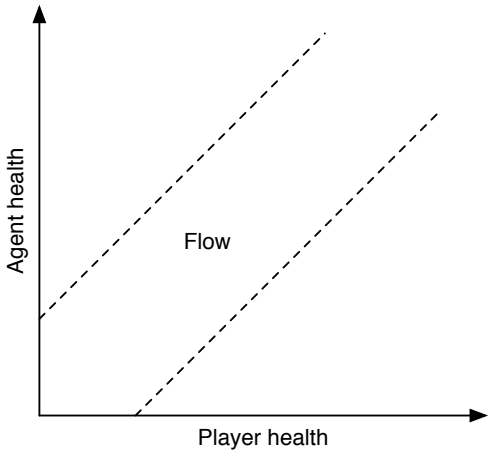


Figure 4.1: Flow channel in our xnaMugen scenario.

Based on Figure 4.1 we consider the player to have the highest chance of being in a state of flow when the health point difference is close to zero. As such, our approaches to DDA will be focused on reducing the health point difference between the players. For both approaches we want to avoid changes in the game mechanics and instead focus on achieving DDA through changes of the agent's behavior. The DDA should not be achieved through means, such as increasing or decreasing health points or damage of attacks.

The secondary objective of our DDA is to make the adjustment of behavior seamless to the player. This should by no means conflict with our first objective, but the approach will be considered better, if it also complies with this objective. By seamless we mean that the agent should be able to adjust its behavior without the player noticing. As such, a

change in behavior should be as transparent to the player as possible making the fight more believable.

4.1 DEFINITIONS

Before we introduce our notion of fitness we will make some basic definitions in relation to our scenario. These definitions will also be used later on in the theory of our DDA approaches.

Player

For our game scenario, we consider the set $Players = \{\rho_1, \rho_2\}$ to be the set of all players. As our scenario only includes two combatants we will not consider any players beyond that.

Health points

In our definition of fitness we include the value of a player's health points. For this purpose we define

$$hp(\rho)_t \mapsto \mathbb{N}$$

where $hp(\rho)_t$ is a function, which given a player $\rho \in Players$ returns the health points at time t.

We also define a constant $\mathcal{H}$, as the maximum amount of health points a player can have, which also equals the amount of health points at time point zero.

Damage

For the purpose of retrieving the damage dealt by a player we define

$$dmg(\rho)_t \mapsto \mathbb{N}$$

as a function which given a player $\rho \in Players$ returns the damage dealt at time t.

4.2 FITNESS

Fitness is often used as a measurement of an individuals ability to survive. In our scenario fitness will describe how well a player performs against an opponent. That is, his ability to defeat the opponent. By measuring the fitness value we can determine how well the player performs against the current challenge level of the game. Based on this, we can adjust the behavior of the agent to even out the difference in health points between the agent and the player. Doing this as the game progresses ensures that the player experiences a balanced game play.

4.2.1 *Fitness Function*

Before we can define a fitness function, we need to look at the different variables in the game and determine which of them are appropriate for describing survivability. As mentioned before, health points describe appropriately how close each player is to defeat. As such, we can look at the difference between the player's health points and conclude that the player with the most health points must be the strongest at the time of observation. Using this knowledge, we define a fitness function $f(\rho)_t$ to measure the fitness of a player at time point t.

$$f(\rho)_t = \frac{hp(\rho)_t - hp(\rho')_t}{\mathcal{H}}$$

where $\rho, \rho' \in Players$ and $\rho \neq \rho'$. The fitness function $f(\rho)_t$ will range from -1 to 1 depending on the performance of the agent. The two extremes of the function are showed below.

If $hp(\rho)_t = 0$, $hp(\rho')_t = 1000$ and $\mathcal{H} = 1000$ then $f(\rho)_t = \frac{0-1000}{1000} = -1$

If $hp(\rho)_t = 1000$, $hp(\rho')_t = 0$ and $\mathcal{H} = 1000$ then $f(\rho)_t = \frac{1000-0}{1000} = 1$

If player ρ is our agent, a fitness value above zero indicates that the agent performs better than the human player. A fitness value below zero will indicate a worse performance than the human player. It is obvious that a positive fitness value for the agent will result in an equally negative fitness for the player and vice versa. This can also be expressed by the following property

$$f(\rho)_t + f(\rho')_t = 0$$

A graphical representation of how the fitness of two players could change over time is illustrated in Figure 4.2.

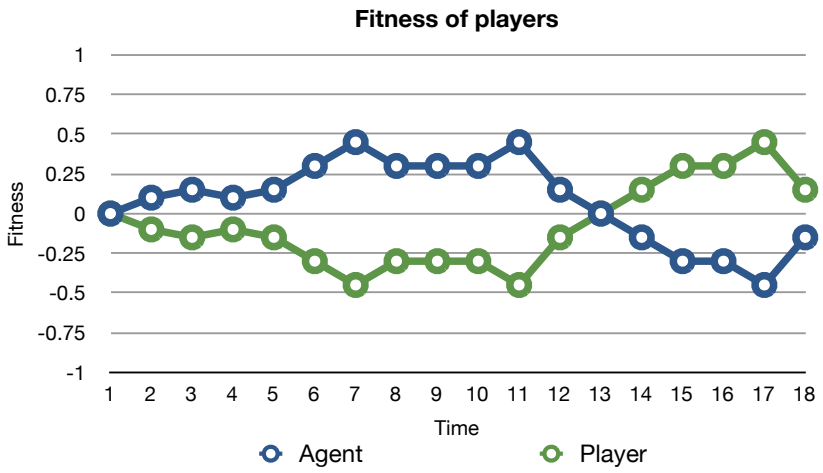


Figure 4.2: The fitness $f(\rho)_t$ of an agent and $f(\rho')_t$ of a player.

For many of our results we will be looking at what we refer to as the AEF. This denotes the average of the End Fitness (EF), the fitness of the agent at the last time step of a match over a series of matches. This measurement will show how great the difference in health points is in average when a match ends. We use this as an indicator of how balanced the matches were on average. If the AEF is very high or very low it indicates that one player had a distinct advantage over the matches. If the AEF is close to zero however, it indicates that most of the matches were even.

5

BEHAVIOR DESIGN

In this chapter we describe the design of our BTs. These are the BTs that will be used in the evaluation of the proposed methods. The BTs designed follow the same template, illustrated in Figure 5.1. The difference in the various BTs is the probability distribution in the *Attack* selector node. The static agents will have a static probability assigned to the *Attack* selector node and the random agents will switch between a set of static behaviors. The DDA agents also follow this BT structure, but the details of the DDA agents will be described in chapter 6 and chapter 7 respectively. In this chapter we often refer to agents, e.g., *B1*, *B2*, etc. This is simply to be understood as an agent utilizing the BT *B1*, *B2*, etc.

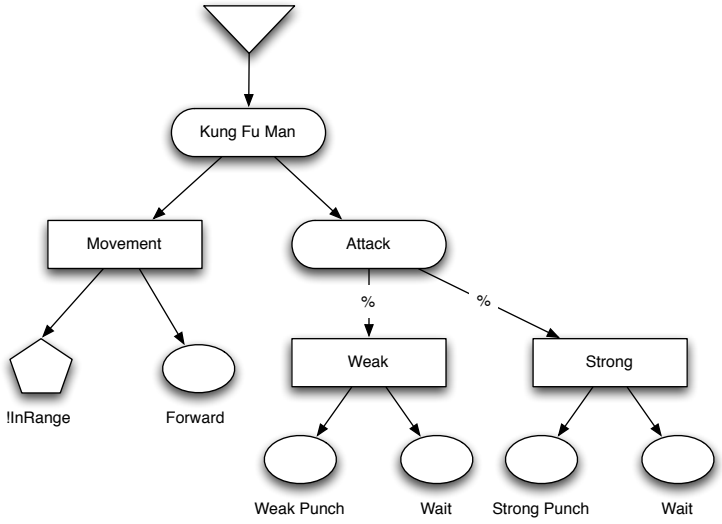


Figure 5.1: BT structure.

According to Figure 5.1 an agent utilizing the BT will move forward, until it is in range of the opponent and when in range either execute a *Weak Punch* or a *Strong Punch*, both with a delay afterwards.

5.1 STATIC BEHAVIORS

We designed two sets with five static BTs each, namely $\mathcal{B}$ and $\mathcal{C}$. The behaviors {B1, B2, B3, B4, B5}$\in \mathcal{B}$ and {C1, C2, C3, C4, C5}$\in \mathcal{C}$. The difference between $\mathcal{B}$ and $\mathcal{C}$ is that the $\mathcal{B}$ behaviors vary more in difficulty as opposed to the $\mathcal{C}$ behaviors which are closer to each other in terms of difficulty. The intention is that the difficulty of the BTs scale from low difficulty, *B1* and *C1*, to high difficulty, *B5* and *C5*. The difficulty of the BT is increased by increasing the probability of executing the *Strong Punch* over the *Weak Punch*. The *Strong Punch* should obviously be stronger than the *Weak Punch* since it has a higher damage output, but to verify the claim we put it to the test.

AI	Win	Loss
Weak Punch AI	0	100
Strong Punch AI	100	0

Table 5.1: Win-loss ratio between *Weak Punch only agent* and *Strong Punch only agent*.

Table 5.1 shows the results of our test, to show that *Strong Punch* is indeed stronger than *Weak Punch*. We designed two agents for this test: *Weak Punch only agent* executes the *Weak Punch* all the time, while the other, *Strong Punch only agent* executes the *Strong Punch* constantly. With 100 rounds and each agent having 2000 health points, we observe the *Strong Punch* agent wins every single time, confirming that *Strong Punch* is the strongest attack among the two.

5.1.1 *Behaviors in $\mathcal{B}$*

The behaviors in $\mathcal{B}$ range from 90%/10% to 10%/90% in the probability distribution in the attack selector node. The agents utilizing these BTs will have a significant difference in difficulty as the probabilities shift with 20% for each behavior. Table 5.2 shows the probability distribution for the BTs.

AI	Weak Punch	Strong Punch
B1	90%	10%
B2	70%	30%
B3	50%	50%
B4	30%	70%
B5	10%	90%

Table 5.2: Probability distribution in the attack selector node for each BT in $\mathcal{B}$.

In order to put some numbers on the difficulty between the BTs, we set them to fight each other to see how they compare. We use the game scenario described in section 2.2 for conducting the tests. What we seek answer to is whether $B1 < B2 < B3 < B4 < B5$ in terms of difficulty. We want to make sure this relation between the agents is true, since it allows us to be certain, whether an agent intended to be stronger actually is stronger. The only way to confirm, whether it is true, is to test the behaviors. In this section we test all static agent against each other.

5.1.1.1 *Evaluation of $\mathcal{B}$ Behaviors*

For each test, there will be a table indicating how the agent performs. Each table consist of five columns: (1) The agent they are fighting against, (2) wins, (3) losses, (4) draws and (5) finally the AEF values. Draws are matches where double knock out occurs, meaning both players get knocked out.

5.1.1.2 *Setup*

The setup for these tests is a 100 matches against each agent, each agent has 2000 hit points, the wait parameter is set to 30 and the range parameter is set to 80 for each agent. We match up agent *B1* against all the other agents including itself to see how it performs. We do the same for agent *B2*, *B3*, *B4* and *B5*.

B1 Performance

B1				
AI	Win	Loss	Draw	AEF
B1	46	43	11	0.0010
B2	0	100	0	-0.1496
B3	0	100	0	-0.2600
B4	0	100	0	-0.3433
B5	0	100	0	-0.4122

Table 5.3: B1 results.

Table 5.3 shows *B1* loses to all the other agents without exceptions and the AEF decreases gradually as it fights against tougher opponents. *B5* wins with approximately 40% of its health left in average. This shows the big gap between *B1* and *B5*. When fighting against itself we see a 46/43 win-loss ratio and an AEF value very close to zero. It is obvious from the results that *B1* is the weakest agent among the $\mathcal{B}$ agents.

B2 Performance

B2				
AI	Win	Loss	Draw	AEF
B1	100	0	0	0.1496
B2	44	51	5	-0.0065
B3	0	98	2	-0.1316
B4	0	100	0	-0.2370
B5	0	100	0	-0.3130

Table 5.4: B2 results.

The results in Table 5.4 show that *B2* was able to beat *B1*, have an equal set of matches against itself and lose to the rest. However, there is one exception, which is that *B2* was able to get two draws against *B3*. Even though the relative difference in strength between them should be clear, and favor *B3*, we still see *B2* getting draw matches. We believe the reason for the few draws is that *B2* executed the *Strong Punch* often despite the low probability. When looking at the AEF value against *B3*, it wins with approximately 13% health points on

average and the overall win-loss ratio still favors *B3*. The AEF values are a bit higher than the ones for *B1* when fighting stronger opponents.

B3 Performance

B3				
AI	Win	Loss	Draw	AEF
B1	100	0	0	0.2600
B2	98	0	2	0.1316
B3	44	43	13	0.0006
B4	0	96	4	-0.1114
B5	0	100	0	-0.2044

Table 5.5: B3 results.

Table 5.5 shows that *B3* wins against *B1* and *B2*, and loses to *B4* and *B5*. We again see the occurrence of a weaker agent, *B2*, being able to get a few draw games against a stronger agent, *B3*, as well as *B3* getting some draws against *B4*. The reasoning for this is the same as for *B2*.

B4 Performance

B4				
AI	Win	Loss	Draw	AEF
B1	100	0	0	0.3433
B2	100	0	0	0.2370
B3	96	0	4	0.1114
B4	36	47	17	-0.0070
B5	0	100	0	-0.2044

Table 5.6: B4 results.

The results in Table 5.6 shows that *B4* has an easy time against *B1*, *B2* and *B3*, while losing against *B5*.

B5 Performance

B5				
AI	Win	Loss	Draw	AEF
B1	100	0	0	0.4122
B2	100	0	0	0.3130
B3	100	0	0	0.2044
B4	100	0	0	0.1030
B5	34	35	31	0.0023

Table 5.7: B5 results.

We see in Table 5.7 that *B5* easily beats all the other agents without exceptions. The AEF values are the opposite of what we saw in the test for *B1*. This makes sense, since *B5* is the strongest agent.

5.1.2 *Behaviors in $\mathcal{C}$*

The $\mathcal{C}$ behaviors slightly resemble the $\mathcal{B}$ behaviors, but differ in that they range from 70%/30% to 30%/70% and shift with 10% for each behavior. The reason for designing the $\mathcal{C}$ behaviors is to have a set of behaviors, where the difference in difficulty is not as large as in the $\mathcal{B}$ behaviors, to verify what impact it will have on the scenario. Table 5.8 shows how the probabilities are distributed for the behaviors.

AI	Weak Punch	Strong Punch
C1	70%	30%
C2	60%	40%
C3	50%	50%
C4	40%	60%
C5	30%	70%

Table 5.8: Probability distribution in the attack selector node for each BT in $\mathcal{C}$.

Similar to the $\mathcal{B}$ behaviors we need to verify the difference between the $\mathcal{C}$ behaviors. This is done by running the same tests on the $\mathcal{C}$ behaviors.

5.1.2.1 *Evaluation of $\mathcal{C}$ Behaviors*

The setup for these tests is the same as for the $\mathcal{B}$ behaviors. The results for the $\mathcal{C}$ agents are more or less the same as for the $\mathcal{B}$ agents. The overall difference is that the fights are closer, as the difference between the $\mathcal{C}$ behaviors is not as significant as for the $\mathcal{B}$ behaviors. All in all the weaker behaviors lose to the stronger behaviors, but still have a few draws against them, as well as a few wins. The AEF values lie closer to each other compared to the previous tests. We refer to Appendix A for details.

6

DYNAMIC DIFFICULTY ADJUSTMENT USING MULTIPLE BEHAVIOR TREES

Our first method of DDA is based on an agent switching between a set of predefined behaviors. This chapter will go through the ideas behind the approach as well as some of the challenges that are present. After that we will present the fundamental theory behind the approach, the proposed algorithm for switching between behaviors and an evaluation of the proposed method.

The inspiration for this approach originates from the game industry, where difficulty is often regarded as a static setting. Usually, when playing a game, the player can switch between a few predefined levels of difficulty. When changing difficulty, the result can be tougher enemies, fewer available resources, etc., depending on the game. The important factor in these approaches is that they all require the player to actively choose an easier or harder difficulty. Thus the player's experience can vary greatly throughout the game, as different parts of a game often vary greatly in difficulty even at the same setting. Our approach correlates to the static difficulty setting known from the industry in the sense that the BTs map to specific difficulty settings. The difference here is that the game itself will change between these predefined difficulty levels in accordance to the player's skill, and does not actively require the player to chose a predefined setting.

In our approach we adjust the behavior of an agent utilizing a set of BTs with different levels of difficulty. The BTs we use for this purpose were described previously in chapter 5. Each of these BTs will function at different fitness intervals. By this we mean that the BT utilized when the agent has a low fitness will be different from the one utilized when the fitness is high. During the fight the agent switches BT as the fitness of the agent changes. For this method to work, the BTs need to vary enough in difficulty, such that both unexperienced and experienced players will find themselves challenged.

The first challenge of this approach is finding the right amount of behaviors needed in the set of available behaviors. With too few behaviors the DDA may become too apparent to the player and the secondary objective to make the DDA seamless will be unfulfilled. With many behaviors the approach may become unfeasible due to the amount of computations and storage needed to handle a larger amount of behaviors. It can also be quite tedious and time demanding to create a large number of behaviors with the appropriate difference in difficulty. Due to this, we decided to utilize the ten different behaviors proposed in chapter 5, with one agent utilizing $B1$, $B2$, $B3$, $B4$ and $B5$, and another utilizing $C1$, $C2$, $C3$, $C4$ and $C5$. Our choice is based on the fact that our primary goal is to create a balanced game with the measurement of fitness defined in section 4.2. To consider the seamlessness in the switch between behaviors would increase the complexity of our approach.

6.1 APPROACH

This approach is based on the notion that each BT is mapped to an interval of the fitness function. When the bounds of an interval is crossed it will indicate the switch from one BT to another. One of the challenges of this approach is to find the intervals which provide a balanced game. Based on these notions, we propose a method for dividing the fitness function into separate intervals for our behaviors. It must be taken into account that the actual interval in which the fitness value fluctuates vary significantly from fight to fight. As such, it would be difficult to predefine intervals before a game. To address this problem we propose a dynamic expansion of the fitness intervals.

Initially we will define an interval as a subset of the entire fitness interval. We will refer to this interval as the Active Fitness Interval (AFI) $\mathcal{A}$ where $\mathcal{A} \subseteq [-1, 1]$. The intervals of our behaviors will be evenly distributed within the AFI top to bottom, from the weakest to the strongest behavior. As the game progresses the fitness of our agent may exceed the boundaries of the AFI. When this happens the AFI will be increased to include the new fitness value and the intervals of all BTs will be recalculated.

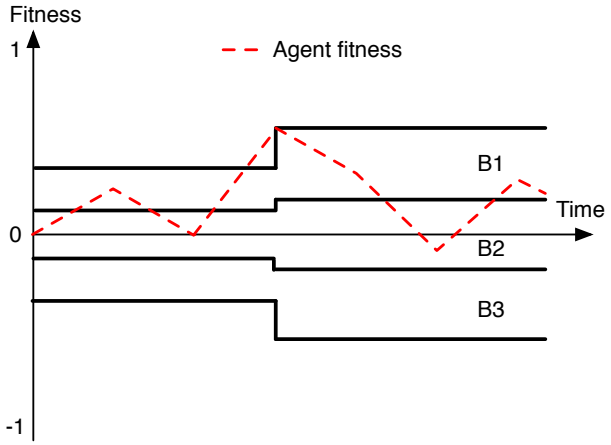


Figure 6.1: Illustration of the intervals of three different BTs.

Figure 6.1 illustrates the AFI of the fitness function and how it is divided into equally sized intervals for the different BTs. The reason for choosing equally sized intervals is to reduce the complexity of the approach. When exceeding either the lower or upper bound of a BT's interval we adjust the difficulty by switching to the BT of the new interval. This approach relies on the assumption that the difficulty of the BTs is ascending from $B1$ to B_n, such that an agent with a high fitness will utilize a weaker BT and vice versa. It is also illustrated how the AFI may change during the game, increasing the size of the respective intervals while still ensuring they are equally sized. Based on this literal description of the relation between the BTs and their intervals we specify a mathematical definition.

Let $\beta = (B, \mathcal{I})$ be a tuple where

$B = \{b_1, b_2, \ldots, b_{n-1}, b_n\}$ is the set of all BTs,

$\mathcal{I}$, where $\forall I \in \mathcal{I}$ s.t. $I \subseteq [-1, 1]$, is the set of all intervals.

We say that

$$\forall(b_i, I_i) \in \beta \text{ and } \forall(b_j, I_j) \in \beta \text{ if } (b_i, I_i) \neq (b_j, I_j) \text{ then } I_i \cap I_j = \emptyset.$$

$$\mathcal{A} \backslash (\bigcup_{i=1}^{|\beta|} I_i) = \emptyset$$

With this we have established that we can have an arbitrary number of BTs, which each are related to an interval of the fitness function. Furthermore, we state that any one interval cannot intersect with the intervals of other BTs. The last definition is to ensure that the union of all intervals equal the AFI to avoid the possibility of having void regions in the AFI. It is important to note that the mathematical definition does not take difficulty into account. As such, there are no rules specifying the order of difficulty for the behaviors. It will be up to the designer of the behaviors to specify the appropriate order.

6.2 ALGORITHM

Based on the theory in the previous section we design an algorithm for adjusting the difficulty of the game based on the fitness value. Furthermore, the algorithm should support the dynamic expansion of the AFI. It is worth noting that the algorithm does not assign thresholds to behaviors based on difficulty, nor does it check whether the threshold of one behavior intersect with the threshold of another. The assumption is that this is already ensured when the set of behaviors is given as input.

Algorithm 1 is the algorithm used for finding the active behavior based on the current fitness value. The algorithm is a two step algorithm where the first step is to check whether the AFI needs to be expanded. The second step of the algorithm is to find the BT corresponding to the current fitness value. At *Line 1*, we check whether an expansion of the AFI is needed; if the fitness value is outside the AFI. If this is the case, the AFI is expanded at *Line 2* using the fitness value to adjust the upper and lower boundary. We make sure that the AFI is expanded equally on both the positive and negative side of the x-axis to get a good distribution of intervals with the stronger behaviors having intervals below the x-axis and the weaker behaviors having intervals above the x-axis.

Based on the new AFI, the algorithm iterates through β and assigns new evenly sized intervals to all behaviors starting from the upper bound of the AFI $\frac{|\mathcal{A}|}{2}$. At *Line 4*, we check whether the actual behavior is the last one in β. If this is not the case, we assign a new interval

$$I_i = \left(\frac{|\mathcal{A}|}{2} - (\frac{|\mathcal{A}|}{|\beta|}(i+1)), \frac{|\mathcal{A}|}{2} - (\frac{|\mathcal{A}|}{|\beta|}i) \right]$$

to the behavior at *Line 5*. For both bounds we calculate $\frac{|\mathcal{A}|}{|\beta|}$, the size of the AFI divided by the number of BTs in β, which is the size of each interval. We find the upper bound of the interval by subtracting the size of the interval multiplied by its position i in β from the upper bound of the AFI $\frac{|\mathcal{A}|}{2}$. As the lower bound of the interval is equal to the upper bound of the next interval in β the same approach can be used to find the lower interval, but with position $i+1$ instead of i. If the behavior is the last in β we assign a fully closed interval to

the behavior at *Line 7*. This ensures that our set of intervals will include every element in the AFI. That the fully closed interval is the last element in β is a choice we made. As the fully closed interval only include one additional element, compared to the other intervals, we claim that this will have no disruptive effect on the approach in general.

The second step of the algorithm, at *Line 8* to *Line 10*, is a search through the set of BTs in β to find the behavior which includes the current fitness value. At *Line 9*, we check whether f is included in I. If this is the case, we return the BT b at *Line 10* and the algorithm terminates.

Algorithm 1: Dynamic difficulty adjustment using dynamic intervals.

Input: A fitness value f, an AFI $\mathcal{A}$ and a tuple $\beta = (B, \mathcal{I})$.
Output: A BT $b \in B$.

```
1  if f ∉ A then
2      A = [−|f|, |f|]
3      for i = 0 to (|β| − 1) do
4          if i ≠ (|β| − 1) then
5              I_i = ( |A|/2 − (|A|/|β|)(i + 1)), |A|/2 − (|A|/|β|)i ) ]
6          else
7              I_i = [ |A|/2 − (|A|/|β|)(i + 1)), |A|/2 − (|A|/|β|)i ) ]
8  foreach (b, I) ∈ β do
9      if f ∈ I then
10         Output b
```

The BT output by the algorithm will be used as the new active BT by the agent making sure that the agent always utilizes a BT with a difficulty corresponding to its fitness.

6.3 RESULTS

We will now look at the tests performed with our first DDA approach. Two agents will be fighting each other, one of which will apply the DDA algorithm while utilizing behaviors from subsection 5.1.1 and subsection 5.1.2. The other agent will be utilizing different static behaviors or switch randomly. Our goal is to test how the DDA agent performs against opponents of varying difficulty. It should be noted that we refer to the agent utilizing the DDA algorithm as *DDA1*. When testing against $\mathcal{B}$ agents *DDA1* will utilize BTs from $\mathcal{B}$. The same holds true for tests conducted against $\mathcal{C}$ agents.

6.3.1 *Setup*

Our tests are conducted over a set of 100 matches with each player having a total of 2000 health points. When playing against the *Random* agent it randomly selects a behavior among the BTs, $\mathcal{B}$ or $\mathcal{C}$, every 5 seconds depending on the test. In all the scenarios the DDA agent will begin the match utilizing either $B3$ or $C3$ as these are the BTs who's interval include the fitness value zero. The following are the initial intervals for the BTs with an initial AFI $=[-0.01, 0.01]$:

B1/C1 INTERVAL: $(0.006, 0.01]$

B2/C2 INTERVAL: $(0.002, 0.006]$

B3/C3 INTERVAL: $(-0.002, 0.002]$

B4/C4 INTERVAL: $(-0.006, -0.002]$

B5/C5 INTERVAL: $[-0.01, -0.006]$

We have chosen a small initial AFI to allow for several expansions the intervals. The intervals will change according to the algorithm as a result of changes in the agent's fitness.

6.3.2 *DDA1 Versus $\mathcal{B}$ Agents*

Table 6.1 and Figure 6.2 show the results of the tests conducted. The fight against *B3* was quite evenly matched with an almost equal amount of wins and losses and a high amount of draws. This changes against *B2* and *B4* where the difference in strength is starting to show. The performance is even worse against *B1* and *B5* where the number of draw games have dropped close to zero.

DDA1			
AI	Win	Loss	Draw
B1	98	0	2
B2	58	8	34
B3	25	23	52
B4	2	58	40
B5	0	91	9
Random	33	37	30

Table 6.1: The results of the *DDA1* against the BTs from $\mathcal{B}$.

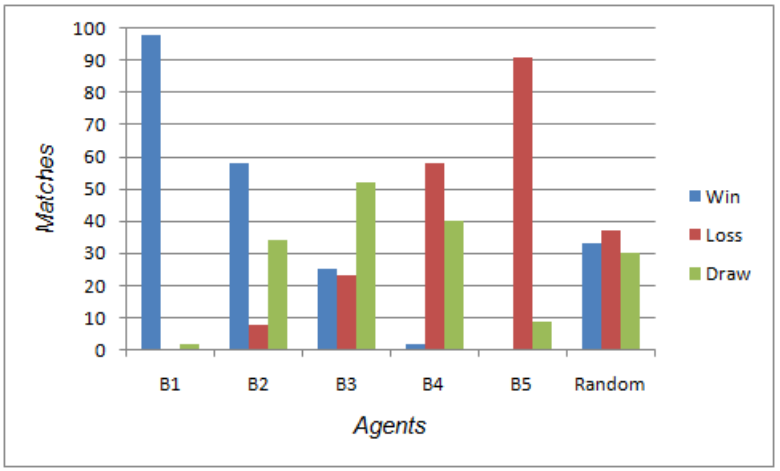


Figure 6.2: Results from *DDA1* versus the BTs from $\mathcal{B}$.

Figure 6.3 shows the health points of our agent and *B4* throughout a match. It is clear that they are very close from start to end.

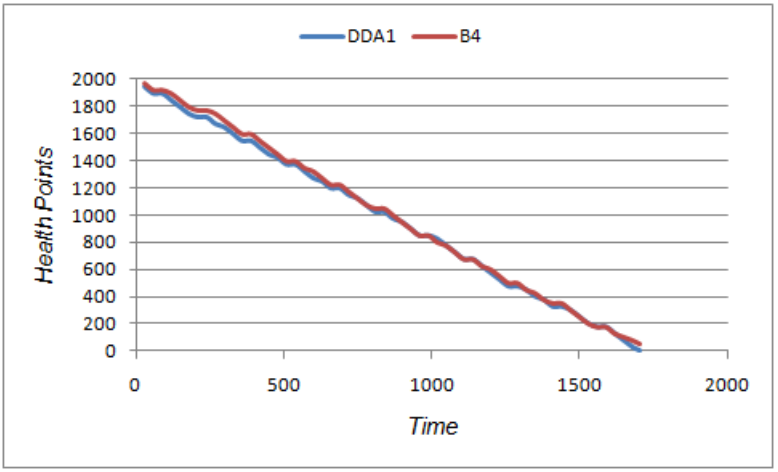


Figure 6.3: Health points through a match between *DDA1* and *B4*.

As the results from all the tests show the same tendency, namely that the health points of both agents is pretty much equal throughout the whole match, we will not elaborate further on this subject. However, what is interesting is the win-loss ratio of the matches. Indeed a balanced game is where the difference in health points of the players fluctuates around zero, but if one always loses he will not have the feeling of a balanced game.

On Figure 6.4 we see how the fitness of our agent changes and how the fitness boundaries increase as the match progresses. The problem with this approach, although the matches are very equal in terms of health points, is that *DDA1* wins a lot against weaker opponents and loses against stronger.

If we look at Figure 6.4, which depicts a match between DDA1 and *B2*, we can find the answer to why the win-loss ratio is as it is. After 300 ticks we see that the intervals have stabilized. The goal is to have the fitness fluctuating around zero, but the problem is that *DDA1* will use *B3*, if the fitness is close to zero. Thus, according to our definition of game balance, we always utilize *B3* in a balanced game scenario. Since *B3* is a stronger behavior than the opponents' we get an advantage. When the fitness of our agent increases we switch to *B2*, the same BT as the opponent. From this point on, if we get further ahead we switch to *B1* only to, eventually, use *B2* again and vice versa for *B3*. To summarize, our agent is forced to use *B2*, the same behavior as the opponent, while having the advantage. This will in the majority of the cases end in a win, some cases in a draw and in very few cases a loss.

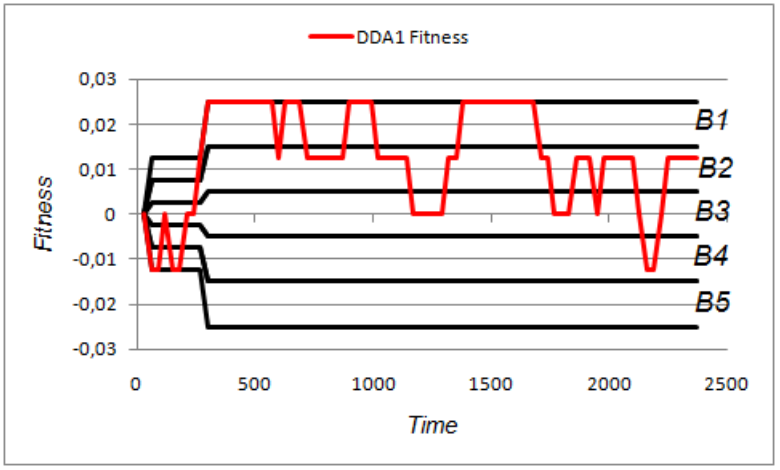


Figure 6.4: A graph that shows when *DDA1* changes behavior against *B2* over the course of a game. The behavior intervals are the areas encapsulated between two lines.

Against *B4* it is the same scenario only below of the x-axis as can be seen on Figure 6.5.

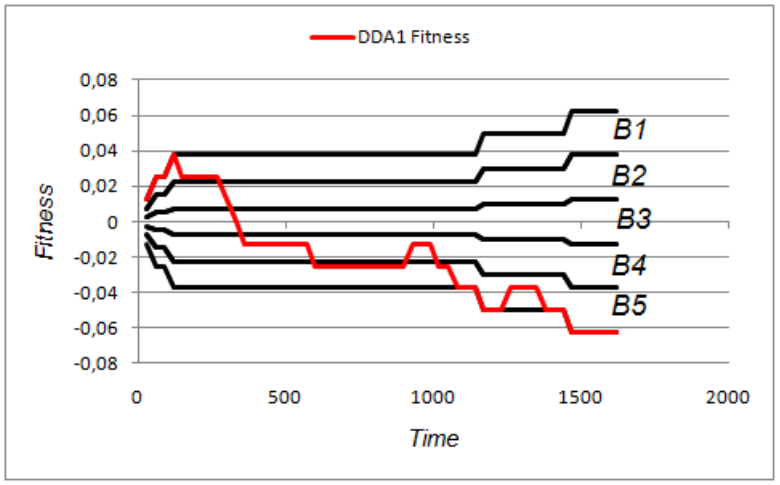


Figure 6.5: A graph that shows when *DDA1* changes behavior against *B4* over the course of a game. The behavior intervals are the areas encapsulated between two lines.

The reason for *DDA1*'s significant number of wins against *B1* and losses against *B5* is that we either start with a stronger behavior, against *B1*, or a weaker behavior againt *B5*. As we cannot use a weaker behavior than *B1* or a stronger than *B5* we are not able to allow *B1* to catch up or catch up to *B5*.

It is apparent from these results that our method has a weakness. When fighting against very strong or very weak behaviors someone will always get a head start. This head start ensures that the agent has an advantage or disadvantage during the entire fight, which in the end results in an imbalanced game. When tested against the *Random* agent the DDA approach showed good results close to those of *B3*. This is likely due to the fact that the probabilities of the *Random* agent converge toward a fifty-fifty probability distribution equal to *B3*.

6.3.3 *DDA1 Versus $\mathcal{C}$ Agents*

From Table 6.2 and Figure 6.6 we see that the overall results of *DDA1* fighting against the agents utilizing $\mathcal{C}$ are not that different from our tests against the BTs in $\mathcal{B}$. The tests against *C3* show the same similarities with an almost identical win-loss ratio. Against agents *C1*, *C2*, *C4* and *C5*, we witness the same results as before with a high ratio of wins against weaker behaviors and a high ratio of losses against stronger behaviors. The tests against the *Random* agent utilizing BTs from $\mathcal{C}$ show a slightly larger ratio of losses opposed to the *Random* agent utilizing $\mathcal{B}$.

	DDA1		
AI	Win	Loss	Draw
C1	91	3	6
C2	77	12	11
C3	40	37	23
C4	12	78	10
C5	2	87	11
Random	29	48	23

Table 6.2: The results of *DDA1* against the BTs from $\mathcal{C}$.

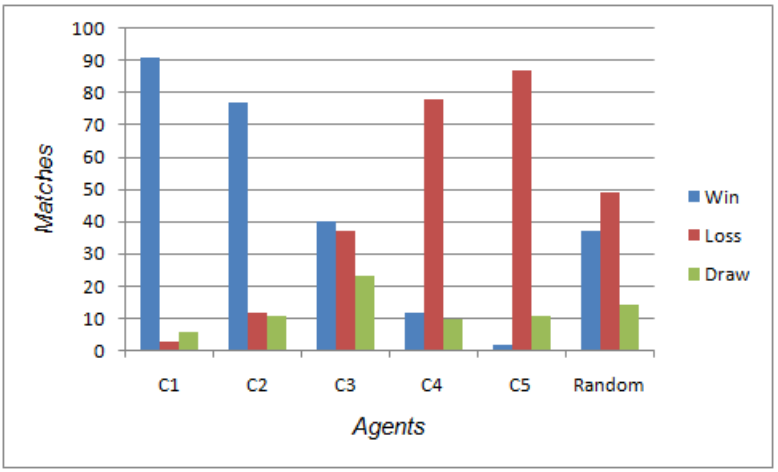


Figure 6.6: Results of *DDA1* against the BTs from $\mathcal{C}$.

6.3.4 *Average End Fitness of DDA1*

In Table 6.3 we see the AEF values for *DDA1*'s matches. The values support the results of the win, loss, draw outcomes, as the AEF values decrease from *B1* to *B5*, going from positive to negative. This corresponds to the large number of wins against *B1* and the large number of losses against *B5*. We also see how a negative AEF corresponds to the cases were the agent lost more matches than it won and vice versa for a positive AEF.

	DDA1	
AI	$\mathcal{B}$	$\mathcal{C}$
1	0.0435	0.0469
2	0.0109	0.0321
3	0.0013	0.0036
4	-0.0145	-0.0305
5	-0.0376	-0.0460
Random	-0.0013	-0.0056

Table 6.3: Comparison of AEF values for *DDA1* against the $\mathcal{B}$ and $\mathcal{C}$ agents.

The small AEF values indicate that the agents have been very close, with regards to the difference between health points, at the end of most fights. This is likely due to the simple scenario we use as a testbed, as it does not allow for much variation in the behavior.

The overall results from the tests against the agents utilizing $\mathcal{B}$ and $\mathcal{C}$ show a clear pattern. When utilizing the same behavior as our opponent from the beginning of a match, we can achieve a high rate of balanced games. However, when there is a difference in difficulty from the beginning, whether the difference is positive or negative, our current approach is not adequate. The agent needs to be more adaptive to its opponent, so initial differences in strength are eventually evened out.

7

DYNAMIC DIFFICULTY ADJUSTMENT USING DYNAMIC BEHAVIOR TREES

In chapter 6 we proposed a DDA method utilizing multiple BTs. While this is a viable approach it requires a lot of work to design several different behaviors. The approach also showed some weaknesses when tested against certain behaviors. This is what inspired us to suggest an approach, which could prove just as effective, while only utilizing one BT. For such an approach to work changes would need to be made to the BT at runtime. This in itself is a challenge, as the behavior of a BT is static in nature and does not change. As such, we would have to come up with an idea of how to introduce these changes.

One suggestion was to make changes to the structure itself at runtime. However, that approach introduces a lot of problems because of the way structural changes may affect the overall goal. Even small changes may have large impacts on the rest of the behavior. This unpredictability made this approach less desirable. The approach we instead chose to explore was the adjustment of parameters in the structure of the BT. While actions themselves may have parameters these can be very game specific and difficult to quantify. For instance, how changes to range and delay may affect the difficulty can be very hard to determine for each and every action available.

Instead of looking at all the different parameters present in the action nodes, we have chosen to focus on the one parameter that is integrated into the framework; the probability distribution on probability selectors. By making adjustments to this probability distribution we can control the outcome of the probability selector and use that to adjust the overall behavior at runtime. This eliminates the need for the different trees we utilized iñ chapter 6 as the probabilities adjust themselves based on the current state of the game.

Before discussing how we adjust the probability distribution we need to introduce some theory. First, we briefly explain a method of optimization called linear programming. We use this method for optimizing the probabilities with respect to the fitness of the agent. Second is the of notion of utilities to quantify the actions of the agent. This is necessary for the definition of the objective function in the linear programming problem, as well as the constraints. For the entire approach we will be using the same BT structure introduced in chapter 5 in Figure 5.1.

7.1 LINEAR PROGRAMMING

Generally speaking *Linear Programming (LP)* is a mathematical method for determining the optimal solution, either by maximizing or minimizing a given mathematical model. It involves a system of linear constraints and a linear function called the *objective function*. The solution is the values of the variables that either minimize or maximize the objective function. More details about LP can be found in Appendix B. It should be noted that the

focus of this project is not to show how to solve a linear programming problem. We utilize a solver as part of the MOSEK framework [5] to solve the problems.

7.2 QUANTIFICATION OF AGENT OUTCOME

How well an agent is performing can depend on several factors. How the performance is measured depends a lot on what the agent's purpose is in the world. For instance, if an agent had a limited power supply and the agent's actions had an energy cost, it would be preferable to get as much done with the minimal use of energy. The behavior of an agent's depends on which actions it executes and when it executes them. This means that the probability distribution on probability selectors have a significant effect on the outcome. In this section we describe the theory behind utility values and how we use utilities in our BTs. The utility theory in this section is based on chapters from [8] and [21].

7.2.1 *Utility*

Utility values are a way of quantifying the outcome of observing some event, making a decision or as in our case the outcome of an agent's actions. When looking at our scenario, one way of measuring the outcome of an agent is to observe how much damage he deals to the opponent. This is a clear indicator of how the agent's actions directly affect the game environment. A rational agent, one that always chooses the best action, would in our scenario always execute the *Strong punch* action. However, it is not our intention to create a rational agent, but rather an agent capable of adapting to a player's skill level. In other words we want to control the damage output of our agent depending on the current state of the game.

$$U(n) \rightarrow \mathbb{R} \text{ where } n \in \textit{Child Nodes} \tag{7.1}$$

In this approach we will assign utilities to each child of the probability selector in order to measure how much damage a specific configuration of the BT outputs. The utility is only meant to represent the damage of executing a child node successfully. It does not in any way extend the notion of BTs. The utility function, Equation 7.1 assigns an integer value to a child.

7.2.2 *Expected Utility*

To calculate the expected utility of an outcome the expected utility function EU, see Equation 7.2, is used. It sums the products of the probabilities of the outcomes and their utility values.

$$EU(\textit{Child Nodes}) = \sum_{n \in \textit{Child Nodes}} P(n) \cdot U(n) \tag{7.2}$$

The utilities on the children will be the damage dealt by successfully executing the respective child node. In our case 25 on the *Weak punch* action and 50 on the *Strong punch* action. Hence, when calculating the expected utility of executing the probability selector we

get the expected average damage our agent will deal when the probability selector samples a child to execute. For instance, if the probability distribution is *0.7* on *Weak punch* and *0.3* on *Strong punch* the expected utility would be $EU = 0.7 \cdot 25 + 0.3 \cdot 50 = 32.5$. This is, at the same time, the average damage output of our agent. In our scenario it is not the goal to achieve maximum utility. Instead we wish to find a probability distribution on the probability selector, s.t. it yields an average damage output that will balance the game as it progresses.

7.3 APPROACH

Before we can optimize the probabilities of the probability selector we need a well defined objective function. Based on our previous notion of fitness defined in section 4.2 the goal is still to minimize the difference in health points between the two agents. Our starting point will be a modification of the function

$$f(\rho)_t = \frac{|hp(\rho)_t - hp(\rho')_t|}{\mathcal{H}}$$

where $\rho, \rho' \in Players$ and $\rho \neq \rho'$. The function $f(\rho)_t$ describes the fitness of player ρ with $hp(\rho)_t$ being the health points of our agent and $hp(\rho')_t$ being the health points of our opponent at time step t. $\mathcal{H}$ denotes the total health points of both players at the beginning of the game. As described in section 4.2 we measure our agent's performance based on the difference in health points at the end of the fight. As such, it makes sense to optimize the EF. Before we move on we need to have a clear definition of how the fitness changes as a result of the damage each player deals.

$$f(\rho)_t = \frac{|(hp(\rho)_t - dmg(\rho')_t) - (hp(\rho')_t - dmg(\rho)_t)|}{\mathcal{H}}$$

where $\rho, \rho' \in Players$ and $\rho \neq \rho'$, with $dmg(\rho)_t$ being the damage dealt by the agent and $dmg(\rho')_t$ being the damage dealt by the opponent at time step t. From this we define a function for the estimated EF. Since we do not know the last time step of the game we will denote this t_{end}. The function for the EF from the current time step t can be defined as

$$f(\rho)_{t_{end}} = \frac{|(hp(\rho)_t - \Sigma_{i=t}^{t_{end}} dmg(\rho')_i) - (hp(\rho')_t - \Sigma_{i=t}^{t_{end}} dmg(\rho)_i)|}{\mathcal{H}}$$

where $\rho, \rho' \in Players$ and $\rho \neq \rho'$. The problem is that we do not know t_{end} or the accumulated damage of either player. It would be possible to find t_{end} for the scenario, where the health points of player ρ is zero, if we knew the accumulated damage of player ρ'. For this reason we make an estimate of the accumulated damage of player ρ'. While we cannot say anything accurate about our opponents damage in the future we can use the past to predict how it might be. We base the prediction on the observations we make of our opponents attacks. As such, the average damage may be defined as

$$dmg_{avg}(\rho)_t = \begin{cases} dmg(\rho) & \text{if } t = 0 \\ \frac{\Sigma_{i=0}^{t} dmg(\rho)_i}{t} & \text{otherwise.} \end{cases}$$

where $dmg_{avg}(\rho)_t$ is the average damage of a player $\rho \in Players$ at time t, $dmg(\rho)_t$ is the damage of a player at time t and $attacks(\rho)_t$ is the number of attacks ρ has executed at time t.

With the estimate of our opponents' average damage we can predict when the match is going to end for player ρ, that is when $hp(\rho) = 0$. This can be calculated by dividing the current health points of our agent with the average damage of our opponent. The result will be the estimated number of remaining time steps assuming there is no change in the opponents' average damage. We will denote the amount of remaining time steps as τ, which can be computed as

$$\tau = \frac{hp(\rho)_t}{dmg_{avg}(\rho')_t}$$

where $\rho, \rho' \in Players$ and $\rho \neq \rho'$. With this we can revise our definition of the estimated EF. Since we are considering a scenario, where player ρ has reached zero health points we can substitute $hp(\rho)_t - \Sigma_{i=t}^{t_{end}} dmg(\rho')_i$ with 0 resulting in the following expression

$$f(\rho)_{t_{end}} = \frac{|0 - (hp(\rho')_t - \Sigma_{i=t}^{t_{end}} dmg(\rho)_i)|}{\mathcal{H}} \tag{7.3}$$

where $\rho, \rho' \in Players$ and $\rho \neq \rho'$. The only unknown term we have left is player ρ's accumulated damage. As we previously know from subsection 7.2.2 our expected damage at a point in time can also be expressed as the expected utility $EU = u_1p_1 + u_2p_2 + \ldots + u_{n-1}p_{n-1} + u_np_n$ of our probability selector. As we have already made the estimate of remaining time steps based on the assumption that our opponent's average damage is constant we do the same here. Using τ, which we calculated earlier we may assume that our accumulated damage can be expressed as $\tau \cdot EU$, which is the accumulated damage over the next τ time steps with an average damage EU in every time step. Substituting with our expression for expected utility from subsection 7.2.2 we can also write this as

$$\tau u_1p_1 + \tau u_2p_2 + \ldots + \tau u_{n-1}p_{n-1} + \tau u_np_n = \tau\Sigma_{i=0}^{n} u_ip_i,$$

where $u_1, u_2, \ldots, u_{n-1}, u_n$ are the utilities of the child nodes. The probabilities $p_1, p_2, \ldots, p_{n-1}, p_n$ are the probabilities of executing each child node respectively, and also the variables we want to find. Substituting this into Equation 7.3 from before and simplifying by removing the parentheses yields the expression

$$f(\rho)_{t_{end}} = \frac{|-hp(\rho')_t + \tau\Sigma_{i=0}^{n} u_ip_i|}{\mathcal{H}}$$

where $\rho, \rho' \in Players$ and $\rho \neq \rho'$, which can also be written as

$$f(\rho)_{t_{end}} = |\frac{1}{\mathcal{H}}\tau\Sigma_{i=0}^{n} u_ip_i - \frac{1}{\mathcal{H}}hp(\rho')_t|$$

where $\rho, \rho' \in \mathit{Players}$ and $\rho \neq \rho'$. Since $\frac{1}{\mathcal{H}}\tau u_1, \frac{1}{\mathcal{H}}\tau u_2, \ldots, \frac{1}{\mathcal{H}}\tau u_{n-1}, \frac{1}{\mathcal{H}}\tau u_n$ are just coefficients and we denote them as $c_1, c_2, \ldots, c_{n-1}, c_n$, and $\frac{1}{\mathcal{H}}hp(\rho')_t$, which is a constant will be denoted k. From this we can derive the objective function we want to optimize

$$|c_1p_1 + c_2p_2 + \ldots + c_{n-1}p_{n-1} + c_np_n - k|$$

where $c_1, c_2, \ldots, c_{n-1}, c_n$ and k are constants and $p_1, p_2, \ldots, p_{n-1}, p_n$ are the unknown probabilities we want to find. The solution to this can be represented as the linear programming problem

$$\mathit{Minimize} \quad |c_1p_1 + c_2p_2 + \ldots + c_{n-1}p_{n-1} + c_np_n - k|$$

subject to a set of constraints. Knowing that $p_1, p_2, \ldots, p_{n-1}, p_n$ are probabilities in a probability distribution places certain constraints on these, as all probabilities must sum to one, and they must all be larger than or equal to zero. This is expressed through the following constraints

$$\begin{aligned} p_1 + p_2 + \ldots + p_{n-1} + p_n &= 1 \\ p_1 &\geq 0 \\ p_2 &\geq 0 \\ &\vdots \\ p_{n-1} &\geq 0 \\ p_n &\geq 0 \end{aligned}$$

The solution to this linear programming problem will be the probability distribution for $p_1, p_2, \ldots, p_{n-1}, p_n$ that yields the fitness value closest to zero at the end of the game based on our estimates at the current point in time t.

7.3.1 *Example*

To make the second approach more comprehensible to the reader we will present an example of its use in our game scenario. Based on the theory presented we will show how to obtain a solution in practice. However, we will not be going into depth with any algorithm for solving linear programming as we utilize the MOSEK framework [5] for this purpose. For the example we will construct an imaginary scenario which could occur in game.

Consider a game at time 30 where the opponent has taken the lead with 900 health points compared to our 800 health points. During this game we have made the following observations of the opponents attacks:

Amount of *Weak punch*: 12

Amount of *Strong punch*: 18

Based on this we calculate our opponents accumulated damage up until the current time step as $\Sigma_0^{30} dmg(\rho') = 1200$. From this we make the following estimate of our opponents future damage at each time step $dmg_{avg}(\rho')_{30} = \frac{1200}{30} = 40$. We use this estimate to predict the number of remaining time steps our agent has left before reaching zero health points. We calculate this as $\tau = \frac{800}{40} = 20$. Now we have all the values we need to estimate the fitness after 50 time steps

$$f(\rho)_{50} = |\frac{1}{2000} \cdot 20 \cdot 25p_1 + \frac{1}{2000} \cdot 20 \cdot 50p_2 - \frac{1}{2000} \cdot 800|$$

This is the expression we use as our objective function. We specify the problem as follows

$$minimize : 0.25p_1 + 0.5p_2 - 0.45$$

subject to

$$\begin{aligned} p_1 + p_2 &= 1 \\ p_1 &\geq 0 \\ p_2 &\geq 0 \end{aligned}$$

Solving this problem using linear programming will result in the following solution

$$\begin{aligned} p_1 &= 0.2 \\ p_2 &= 0.8 \end{aligned}$$

These are the probabilities we now assign to our probability selector. The probability of choosing weak punch will be equal to p_1 and the probability of choosing strong punch will be p_2. If we calculate the observed probability distribution of our opponents attacks we get

$$\begin{aligned} p_1 &= \frac{19}{25} = 0.4 \\ p_2 &= \frac{6}{25} = 0.6 \end{aligned}$$

Compared to our new distribution we can see how we get a slightly higher probability of choosing a the stronger attack to even out the difference between our agent and the opponent.

7.4 RESULTS

We will now look at the tests performed with our second DDA approach. The agent utilizing the DDA method is referred to as *DDA2*. *DDA2* will be evaluated against an agent utilizing $\mathcal{B}$ or $\mathcal{C}$ as well as a *Random* behavior.

7.4.1 *Setup*

Our tests are conducted over a set of 100 matches with each player having a total of 2000 health points. When playing against the *Random* agent it randomly selects a behavior among the BTs, $\mathcal{B}$ or $\mathcal{C}$, every 5 seconds depending on the test. The probability distribution over the attacks *Strong Punch* and *Weak Punch* for *DDA2* will start out with as a uniform distribution. We optimize the probability distribution for the two attacks every second in the game.

7.4.2 *DDA2 Versus B Agents*

Figure 7.5 and Table 7.1 show some good results. Looking at the static agents we see they all have around 50 draw games, which indicates that half of the fights have been fully balanced. Observing the ratio of won games we see how it decreases gradually when fighting tougher opponents. As for the ratio of lost games we see the opposite. Against the *Random* agent we see a draw ratio below that of the static agents. We believe the reason to be the rapid switching between behaviors. *DDA2* has to adapt to the new behavior every 5 seconds when the *Random* agent switches, always leaving it a step behind.

	DDA2		
AI	Win	Loss	Draw
B1	39	11	50
B2	26	28	46
B3	22	29	49
B4	19	32	49
B5	7	41	52
Random	32	42	26

Table 7.1: Results for matches between *DDA2* and the $\mathcal{B}$ agents.

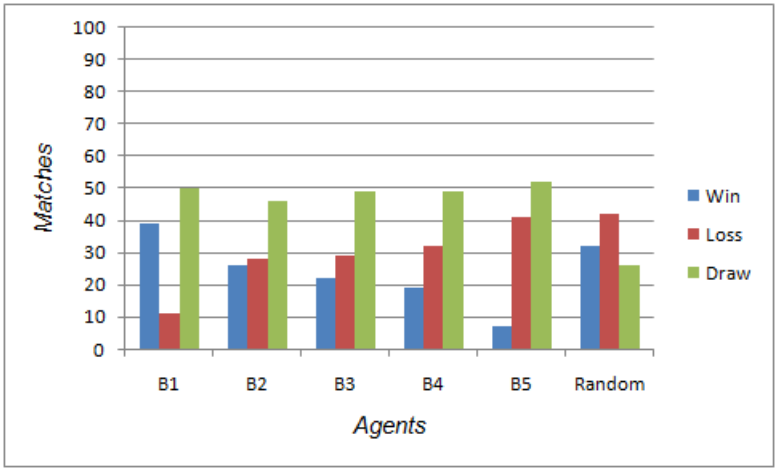


Figure 7.1: Results for matches between *DDA2* and the $\mathcal{B}$ agents.

We chose to take a closer look at a fight between *DDA2* and *B5* to find an explanation for the continuous decrease in won games. Figure 7.2 shows that it is a very equal fight with small fluctuations of the fitness. None of the agents ever attain more than 0.0125 fitness which translates to a difference in health points of $0.0125 \cdot \mathcal{H} = 25$. *B5* was the winner of this match.

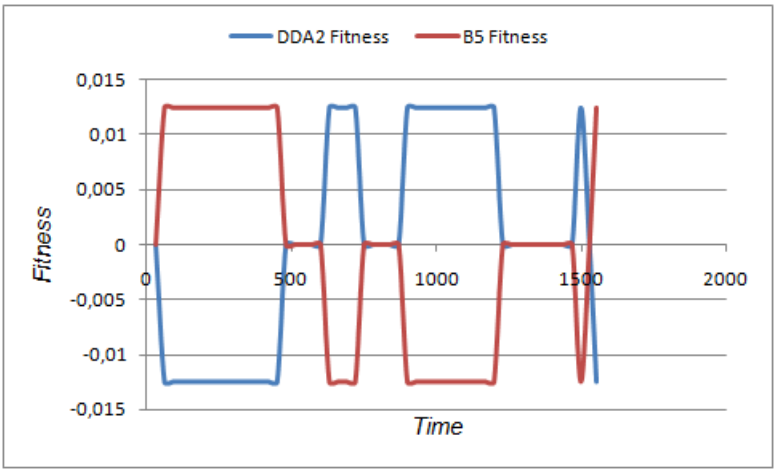


Figure 7.2: Fitness values in a *DDA2* versus *B5* fight.

Figure 7.3 supplements the fitness chart by showing the actual health point values for the agents, following each other closely throughout the game.

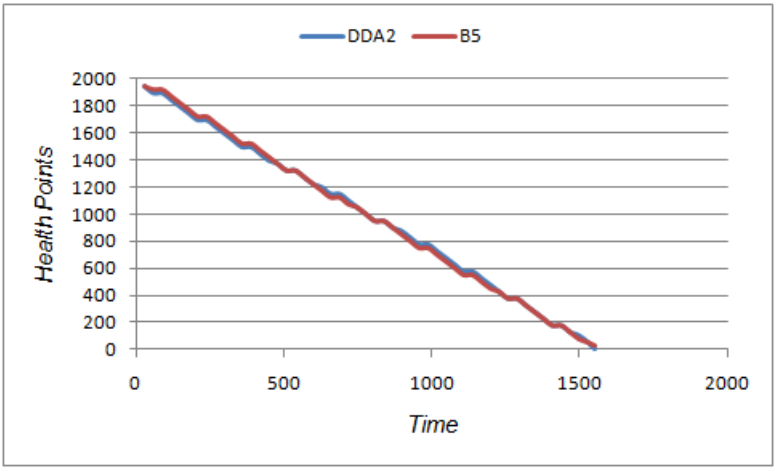


Figure 7.3: Health points in a *DDA2* versus *B5* fight.

Figure 7.4 shows the probability distribution over the attacks of the two agents. *B5* has a high probability of executing the strong attack and does so a lot in the beginning of the fight. *DDA2* updates its probability distribution to resemble this. As the game progresses the observed distribution of *B5*'s attacks converges toward the actual values, namely 0.1 for *Weak punch* and 0.9 for *Strong punch*. The probabilities of the two agents' attacks follow each other but at the end of the fight *DDA2* shifts to a 0.6/0.4 distribution for *Weak punch* and *Strong punch* respectively. We believe the reason for this is that *DDA2* attempts to achieve a draw game and by doing so lowers the probability for the strong punch to avoid knocking

B5 out too early. As *B5* has a high probability of executing the strong attack the match will most likely end in a win for *B5*, but as *DDA2* still has a reasonable chance of executing the strong attack it can also end in a draw.

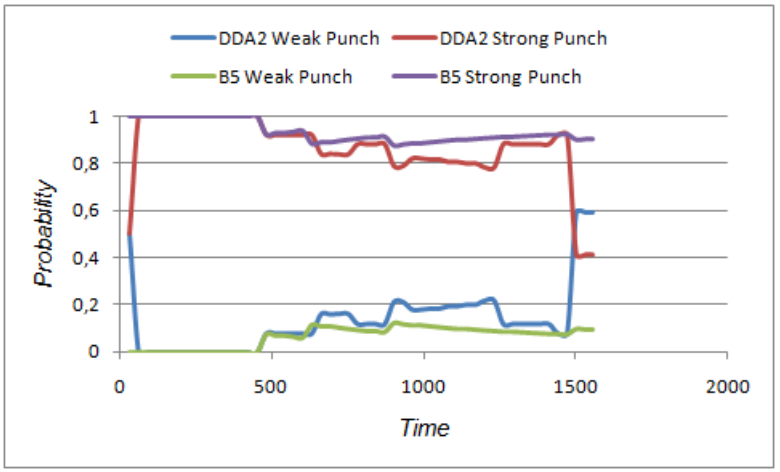


Figure 7.4: Probability distribution of attacks in a *DDA2* versus *B5* fight.

It more or less comes down to chance when looking at the outcome. First, in this particular game both agents got down to 50 health points at the end of the fight. If we assume this is the case in most of the matches that are very close, the probabilities for certain outcomes are the following:

Games ending with a draw is occurs when both players execute the *Strong punch* or the *Weak punch* at the same time. The probability of both players executing a *Strong attack* is $0.4 \cdot 0.9 = 0.36$. The probability of both players executing a *Weak attack* is $0.6 \cdot 0.1 = 0.06$. Thus, the probability of *DDA2* winning is, if he executes a *Strong punch* and *B5* executes a *Weak punch*, $0.4 \cdot 0.1 = 0.04$. Similarly, the probability of *B5* winning is $0.6 \cdot 0.9 = 0.54$.

Although this is only an assumption, the numbers correspond fairly well with the actual outcome of the 100 matches between *DDA2* and *B5*, 7 wins, 41 loses and 52 draws.

7.4.3 *DDA2 Versus C Agents*

To support the results from subsection 7.4.2 we tested *DDA2* against the agents from $\mathcal{C}$ as well as a *Random* agent utilizing the BTs from $\mathcal{C}$. The setup is the same as in subsection 7.4.2. 100 matches with each agent having 2000 health points. 30 in delay and 80 in range. The *Random* agent will be switching BT every 5 seconds and our *DDA2* agent will optimize its probability distribution every second.

Figure 7.5 and Table 7.2 supplements the results from subsection 7.4.2. The high amount of draws indicates the matches have been balanced. Furthermore, we see an approximately equal win-loss ratio for all the agents. For the fights against the *Random* agent we see balanced results with a good win-loss ratio and many draw games.

	DDA2		
AI	Win	Loss	Draw
C1	25	21	54
C2	27	21	52
C3	19	27	54
C4	23	28	49
C5	13	26	61
Random	29	28	43

Table 7.2: Results from *DDA2* versus the $\mathcal{C}$ agents.

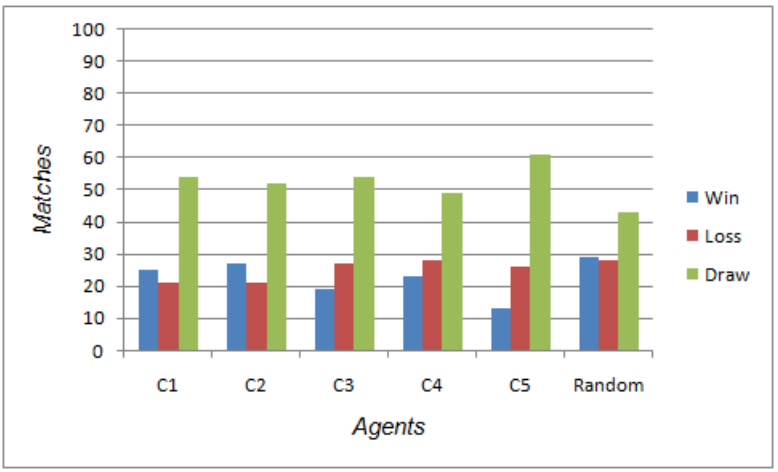


Figure 7.5: Results from *DDA2* versus the $\mathcal{C}$ agents.

7.4.4 *Average End Fitness of DDA2*

In Table 7.3 we see the AEF values for *DDA2*'s matches. The results further underline the outcome discussed previously, as the AEF values are almost equal to each other, and very close to zero. This is mostly due to the large number of draw games as a draw results in an EF of zero. Looking at the actual values we see how a negative AEF corresponds to the cases were the agent lost more matches than it won and vice versa for a positive AEF.

	DDA2	
AI	$\mathcal{B}$	$\mathcal{C}$
1	0.0018	0.0001
2	-0.0002	0.0011
3	-0.0039	-0.0038
4	-0.0050	-0.0030
5	-0.0083	-0.0040
Random	-0.0081	-0.0016

Table 7.3: Comparison of AEF values for *DDA2* against the $\mathcal{B}$ and $\mathcal{C}$ agents.

We believe that a more complex scenario could have resulted in larger AEF values. Due to the simplicity of the scenario, with only two attacks being available, the agents will follow each other closely throughout the match, always ending with a minuscule difference in health points.

7.4.5 *Summary*

The overall results from the tests conducted against the $\mathcal{B}$ and $\mathcal{C}$ agents have been promising. All the scenarios have a high ratio of draw games, approximately half of the games. This combined with an overall good ratio between wins and losses show the feasibility of the *DDA2* approach. We have shown how the optimization of the probability distribution on a probability selector quickly converges to the distribution utilized by the opponent. This illustrates the adaptability of *DDA2*, a property we found lacking in our first approach utilizing *DDA1*.

Appendix C shows a comparison of the results for *DDA1* and *DDA2*. These results further support the claim that *DDA2* is overall more capable of ensuring a balanced game than *DDA1*. *DDA1* is only able to create a fully balanced gameplay in those scenarios where it utilizes the same behavior as the opponent, such as B_3, C_3, and the *Random* agents. *DDA2* achieves to a create balanced game play against all its opponents and we see how *DDA2* adapts well to its opponents in all the test scenarios.

While both our approaches are only proofs of concepts, in the sense that they are applied to a simple scenario, we believe that DDA using dynamic BTs holds the most potential of our two approaches. Even though improvements could be made to both approaches the benefit of working with only one BT instead of several is hard to overlook. Though it is difficult to predict whether the *DDA2* approach, or variation thereof, could be applicable in commercial games, we believe our intention of proposing a valid approach to DDA using BTs have been successful.

8 FUTURE WORK

In this report we have shown how DDA can successfully be applied in a 2D fighting game scenario using BTs. Even though the results for our scenario were promising there are still a lot of room for improvements and further research. We will summarize some of the improvements which we considered for DDA using dynamic BTs, but were not within the scope of this project. For future work we will consider improvements to our DDA approach using dynamic BTs as this was our most successful method.

During the work done on DDA our scenario has been simple. This was necessary to produce viable results in the short time span we had. However, if one was to look at BTs in video games they would find the complexity magnitudes larger. It would be interesting for us to apply our second approach to a large scale scenario and test its feasibility. This could both serve as further evidence for the feasibility of the approach or reveal new problems which could require adjustments to the approach.

Applying the approach to a large scale scenario would also allow us to investigate the scalability of the approach. In terms of scalability it would be interesting to investigate the effect an increased number of children, of a single probability selector, will have on the performance of the approach. Performance could be tested both in terms of computational complexity and variation of the probability distribution on the child nodes. This could be expanded with an investigation of the effect several probability selectors, all with adjustment of their probabilities, would have on performance. The notion of several dynamic probability selectors also raises other issues as it would require adjustment of a hierarchy of probability selectors, some of which rely on others.

While working on our approach we also discovered some interesting problems with regards to the quantification of actions using utilities. As BTs do not inherently support utilities, adding utilities to the framework proved to be a difficult task. It could be interesting to research whether the addition of utilities to BTs is possible, and if there would be a use for it. If possible, this could be a proposed extension to the standard BT framework.

When considering that our approach has only been applied to a 2D fighting game another step could be to test the feasibility of the approach in other game genres. One genre in particular is the FPS game genre where BTs have already been used in several major titles [13, 18]. FPS could also provide an interesting platform for more complex scenarios with multiple actors and 3-dimensional environments.

Another way of developing upon the approach could be to further increase the complexity of the fitness function, by adding additional variables to the function which take into account more varied aspects of the game play. It would be interesting to investigate whether it is possible to include aspects, such as the play style of the player. This could allow for a categorization of him based on, for instance, his level of combat expertise and attack

preferences. To give an example, players who are not using advanced moves such as combos will not be matched against agents frequently utilizing advanced moves. On the other hand, advanced players who are familiar with the advanced attack moves will be matched up against opponents using similar attacks. This could serve as a great basis for diverse agent behavior based on observations made of the human player.

The idea of observing the human player and use it for the creation of a player model is also one we have considered. Currently our only observation of the player, is the observation of his attacks. It is possible that a more complex model of the player including more aspects of his behavior could provide interesting results. There already exist several techniques for player modeling. Combining some of these with our DDA approach could reveal some interesting possibilities for more varying DDA based on a player's preferences in a game scenario.

Finally, when considering our test results, they are all based on synthetic agents as test subjects. It could be interesting to cross verify these results using human test subjects to see whether the results will stay the same or be significantly different. As the game scenario is currently quite simple this would be more interesting in a larger scale scenario.

9 CONCLUSION

We introduced two different methods of applyingDDA in a 2D fighting game. The purpose of both approaches was to show how DDA may be applied to a simple game scenario using BTs for modeling the behavior of the actors.

As preliminary work we reviewed several other approaches on the topic of DDA and player experience. Drawing inspiration from this research we have discussed the concept of DDA in relation to the *flow model* and defined game balance in our own scenario. We introduced a fitness function based on health of both players which we found to be an accurate measurement of the challenge level of the game. Based on the fitness function we described the circumstances under which we considered the game to be balanced. For the purpose of testing we designed a set of BTs specifically suited for our scenario. Tests were conducted to assess the difference in difficulty between the BTs.

The first approach draws inspiration from the game industry, which favors predefined behavior, and introduces a way of switching between BTs of varying difficulty based on the fitness of an agent. An algorithm was designed for the purpose of switching behavior when certain conditions were met. Tests were conducted against opponents utilizing predefined BTs. The results were insufficient in ensuring a balanced game and also revealed some weaknesses in the approach. The biggest of these weaknesses was its inability to adapt when facing some of the predefined BTs. These weaknesses led us to consider a second approach with only one BT.

The second approach introduces a way to adjust the difficulty of a BT by changing the probabilities in the BT structure dynamically. An objective function to be minimized was specified based on the difference between the health of the players. To achieve this, we introduced a notion of utilities on children of the probability selectors in the BT. Based on an estimate of our opponents' performance and the utilities we made the necessary adjustments to our agents' behavior. Tests conducted show promising results for this approach as the game proved to be balanced against all the opponents utilizing predefined BTs. The agent was capable of adapting to opponents quickly and retain a balanced game for the remainder of the match.

Our results of the tests conducted for the two approaches show how the success of the first approach was very situational and the time spent designing several behaviors would be better avoided. Adjusting the probabilities in the BT provided a more adaptable approach to DDA while utilizing only one BT.

DDA in 2D fighting games by adjusting probabilities in a BT structure is certainly a possibility. However, we cannot say whether it is feasible in large scale scenarios as our testing scenario was very specific and only for the purpose of demonstrating the proposed ideas. Further research and testing would have to be conducted in order to provide more

evidence for the generality of the approach in a large scale scenario, i.e., a modern video game. We believe that the approach has shown its potential and with further research it could prove feasible in other game genres.

APPENDIX A

C1				
AI	Win	Loss	Draw	AEF
C1	52	42	6	0.0042
C2	11	81	8	-0.075
C3	1	98	1	-0.1213
C4	0	100	0	-0.1875
C5	0	100	0	-0.231

(a) *C1* results

C2				
AI	Win	Loss	Draw	AEF
C1	81	11	8	0.075
C2	46	44	10	0.0025
C3	12	78	10	-0.0559
C4	2	94	4	-0.1149
C5	0	100	0	-0.17

(b) *C2* results

Table A.1: *C1* (a) and *C2* (b) results.

C3				
AI	Win	Loss	Draw	AEF
C1	98	1	1	0.1213
C2	78	12	10	0.0559
C3	57	36	7	0.0141
C4	14	79	7	-0.051
C5	0	98	2	-0.1109

(a) *C3* results

C4				
AI	Win	Loss	Draw	AEF
C1	100	0	0	0.1857
C2	94	2	4	0.1149
C3	79	14	7	0.051
C4	42	44	14	0.0005
C5	10	79	11	-0.0561

(b) *C4* results

Table A.2: *C3* (a) and *C4* (b) results.

C5				
AI	Win	Loss	Draw	AEF
C1	100	0	0	0.231
C2	100	0	0	0.17
C3	98	0	2	0.1109
C4	79	10	11	0.0561
C5	43	41	16	0.0029

Table A.3: *C5* results.

B

Definition B.1 –

Given $b = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}$ *in* $\mathbb{R}^m$, $c = \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix}$ *in* $\mathbb{R}^n$ *and an* m x n *matrix* $A = [a_{ij}]$,

the CANONICAL LINEAR PROGRAMMING PROBLEM *is the following:*

Find an n-tuple $x = \begin{bmatrix} x_1 \\ \vdots \\ b_n \end{bmatrix}$ *in* $\mathbb{R}^n$ *to maximize*

$$f(x_1, ..., x_n) = c_1x_1 + c_2x_2 + \cdots + c_nx_n$$

subject to the constraints

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &\leq b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &\leq b_m \end{aligned}$$

and

$$x_j \geq 0 for j = 1, ..., n$$

This may be restated in vector-matrix notation as follows:

$$\text{Maximize } f(x) = c^T x \tag{B.1}$$

$$\text{subject to the constraints } Ax \leq b \tag{B.2}$$

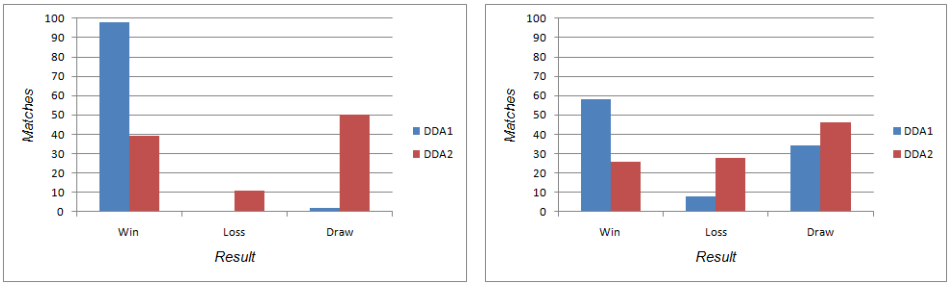
$$\text{and } x \geq 0 \tag{B.3}$$

where an inequality between two vectors applies to each of their coordinates.

Any vector x *that satisfies (6.2) and (6.3) is called a* FEASIBLE SOLUTION, *and the set of all feasible solutions, denoted by* $\mathcal{F}$, *is called the* FEASIBLE SET. *A vector* $\overline{x}$ *in* $\mathcal{F}$ *is an* OPTIMAL SOLUTION *if* $f(\overline{x}) = max_{x \epsilon \mathcal{F}} f(x)$.

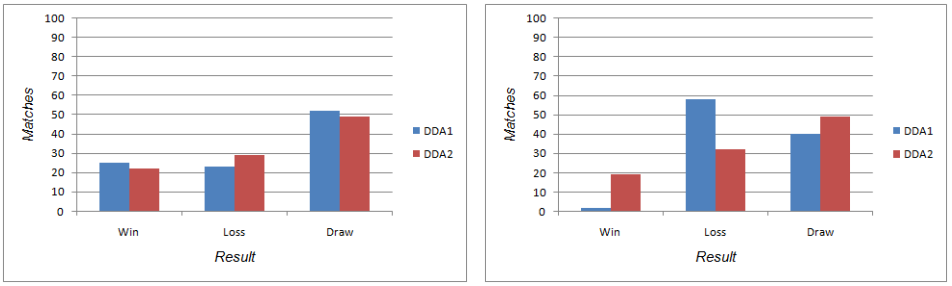
In more general terms x represents the vector of variables we want to determine, c and b are vectors of known coefficients and A is a known matrix of coefficients. The objective function we want to maximize or minimize is $f(x) = c^T x$. And lastly the equations $Ax \leq b$ are the constraints over which the objective function is to be optimized. We will not delve into further details with linear programming, but if the reader is interested we will refer to Chapter 9 of [14].

APPENDIX C



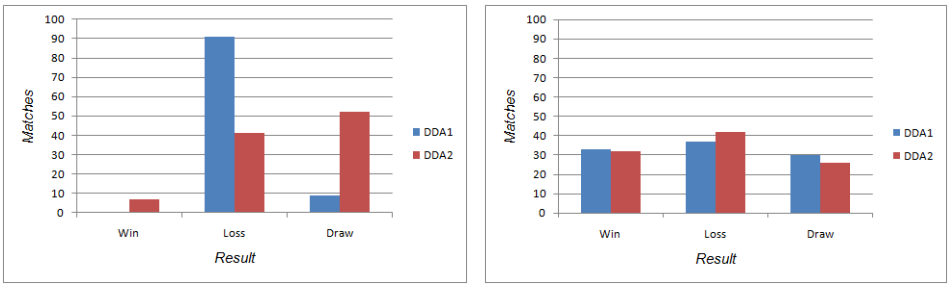
(a) (b)

Figure C.1: Comparison of results for *DDA1* and *DDA2* against *B1* (a) and *B2* (b).



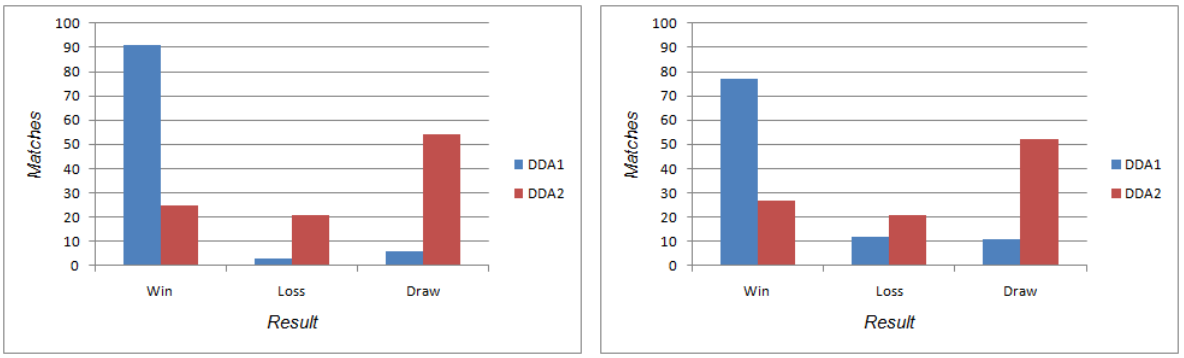
(a) (b)

Figure C.2: Comparison of results for *DDA1* and *DDA2* against *B3* (a) and *B4* (b).



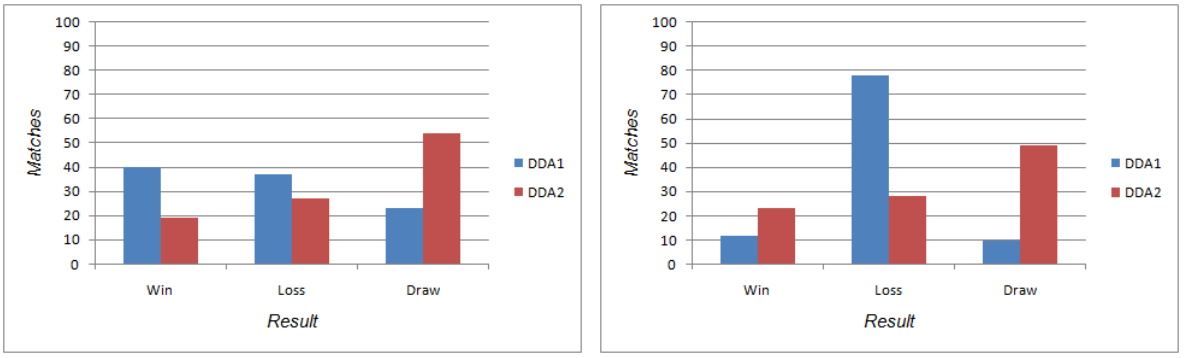
(a) (b)

Figure C.3: Comparison of results for *DDA1* and *DDA2* against *B5* (a) and *Random* $\mathcal{B}$ (b).



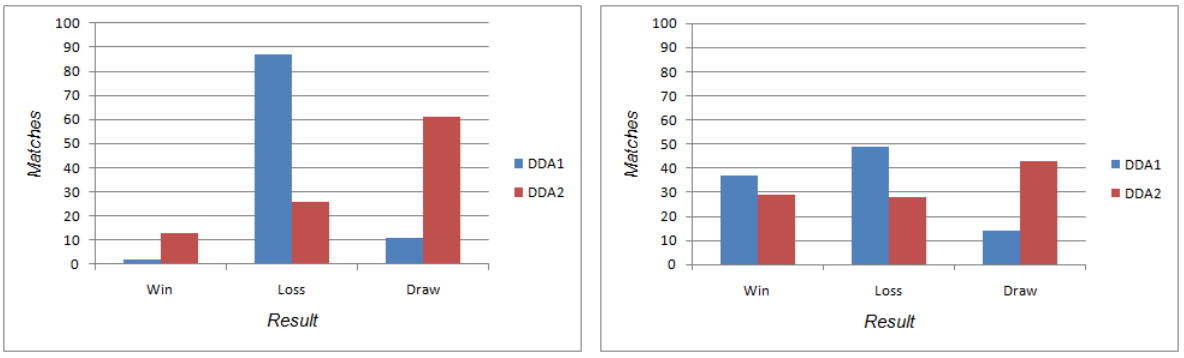
(a) (b)

Figure C.4: Comparison of results for *DDA1* and *DDA2* against *C1* (a) and *C2* (b).



(a) (b)

Figure C.5: Comparison of results for *DDA1* and *DDA2* against *C3* (a) and *C4* (b).



(a) (b)

Figure C.6: Comparison of results for *DDA1* and *DDA2* against *C5* (a) and *Random* $\mathcal{B}$ (b).

BIBLIOGRAPHY

[1] *Flow model image*. http://www.the-happy-manager.com/image-files/self-motivation-flow-model.png. Downloaded: 17.05.2011.

[2] G. D. Andrade, H. P. Santana, A. W. B. Furtado, Ándre Roberto Gouveia Do Amaral Leitão, and G. L. Ramalho. Online Adaptation of Computer Games Agents: A Reinforcement Learning Approach. In *II Workshop de Jogos e Entretenimento Digital*, pages 105–112, 2003.

[3] G. D. Andrade, H. P. Santana, G. L. Ramalho, and V. Corruble. Extending Reinforcement Learning to Provide Dynamic Game Balancing. In *Proceedings of the 2005 IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games*, pages 7–12, 2005.

[4] M.-V. Aponte, G. Levieux, and S. Natkin. Scaling the Level of Difficulty in Single Player Video Games. In *Proceedings of the 8th International Conference on Entertainment Computing*, ICEC '09, pages 24–35, 2009.

[5] M. ApS. *The MOSEK Optimization Software*. http://www.mosek.com, 2011. Downloaded: 24.04.2011.

[6] M. Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. Number ISBN: 978-0060920432 in First Edition. Harper Perennial, 1991.

[7] Elecbyte. *M.U.G.E.N*. http://www.elecbyte.com/mugen, 2011. downloaded: 07.03.2011.

[8] T. D. N. Finn V. Jensen. *Bayesian Networks and Decision Graphs*. Number ISBN: 0-387-68281-3 in Second edition. Springer, 2007.

[9] G. Flórez-Puga, M. A. Gómez-Martín, P. P. Gómez-Martín, B. Díaz-Agudo, and P. A. González-Calero. Query-Enabled Behavior Trees. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(4):298–308, 2009.

[10] Frantzx. *xnaMugen*. http://code.google.com/p/xnamugen/, 2011. Downloaded: 09.05.2011.

[11] J. Hagelbäck and S. J. Johansson. Measuring player experience on runtime dynamic difficulty scaling in an RTS game. In *Proceedings of the 5th international conference on Computational Intelligence and Games*, CIG'09, pages 46–52, 2009.

[12] R. Hunicke and V. Chapman. AI for Dynamic Difficulty Adjustment in Games. In *Challenges in Game Artificial Intelligence AAAI Workshop*, pages 91–96, 2004.

[13] D. Isla. Handling Complexity in the Halo 2 AI. http://www.gamasutra.com/gdc2005/features/20050311/isla_01.shtml (checked 08.06.2011), 2005.

[14] D. C. Lay. *Linear Algebra and Its Applications*. Number ISBN: 0-321-31485-9 in Third Edition. Pearson Education, 2006.

[15] Midway Games. Mortal Kombat. http://en.wikipedia.org/wiki/Mortal_Kombat (checked 08.06.2011), 1992.

[16] Namco. Tekken. http://en.wikipedia.org/wiki/Tekken (checked 08.06.2011), 1994.

[17] J. K. Olesen, G. N. Yannakakis, and J. Hallam. Real-time challenge balance in an RTS game using rtNEAT. In *IEEE Symposium On Computational Intelligence and Games, CIG'08.*, pages 87–94, 2008.

[18] R. Pillosu. Coordinating Agents with Behavior Trees. http://staff.science.uva.nl/~aldersho/GameProgramming/Papers/Coordinating_Agents_with_Behaviour_Trees.pdf (checked 08.06.2011), 2009.

[19] S. Rabin. *AI Game Programming Wisdom 4*. Number ISBN: 978-1-58450-523-5 in First Edition. Charles River Media, 2008.

[20] P. Spronck, I. Sprinkhuizen-Kuyper, and E. Postma. Difficulty Scaling of Game AI. In *Proceedings of the 5th International Conference on Intelligent Games and Simulation (GAME-ON 2004)*, pages 33–37, 2004.

[21] P. N. Stuart Russell. *Artificial Intelligence - A Modern Approach*. Number ISBN: 0-13-207148-7 in Third Edition. Pearson, 2010.

[22] Team Ninja. Dead or Alive. http://en.wikipedia.org/wiki/Dead_or_Alive_(series) (checked 08.06.2011), 1996.

[23] J. Tolentino. *Good idea, Bad idea: Dynamic Difficulty Adjustment*. http://www.destructoid.com/good-idea-bad-idea-dynamic-difficulty-adjustment-70591.phtml (checked 08.06.2011), 2008.

[24] Valve. *Half-Life 2*. http://www.valvesoftware.com/games/hl2.html (checked 08.06.2011), 2004.

[25] G. van Lankveld, P. Spronck, J. van den Henrik, and M. Rauterberg. Incongruity-Based Adaptive Game Balancing. In *Advances in Computer Games*, volume 6048 of *Lecture Notes in Computer Science*, pages 208–220. Springer, 2010.

[26] M. A. Verma and P. W. McOwan. An adaptive methodology for synthesising Mobile Phone Games using Genetic Algorithms. In *The 2005 IEEE Congress on Evolutionary Computation*, volume 1, pages 864–871, 2005.