Exploring learning techniques based on decision trees and their performance in platform games

Final Degree Work

Author: Carlos Tello Ordoñez
Bachelor’s Degree in Video Game Design and Development
Universitat Jaume I
July 6th, 2020

Advisor: Pedro José Sanz Valero

ABSTRACT

This document presents the Final Degree Work of the Bachelor's Degree in Video Game Design and Development. The work consists of the study and implementation of machine learning techniques based on decision trees. The focus is set on Quinlan's Inductive Decision Tree algorithm (ID3) [1] and its extension, the Incremental Decision Tree learning algorithm (ID4).

The learning methods are applied to the classic Super Mario Bros [2]. The artificial intelligence agents are implemented and trained within the Mario AI Framework [3]. This is a framework for using AI methods with a version of Super Mario Bros. The framework includes features such as level generators, observation grid, and already implemented playing agents.

In order to demonstrate the reliability and feasibility of the system, some tests have been carried out as an experimental validation. These preliminary results showcase the pros and cons of the applied learning approach and open the door to continue exploring learning techniques in other videogame contexts.

KEYWORDS

- Machine learning.
- Induction.
- Decision Trees.
- Incremental learning.
- Platform games.

CONTENTS

ABSTRACT 1

KEYWORDS 1

1. TECHNICAL PROPOSAL 5

1.1. INTRODUCTION AND MOTIVATION OF THE PROJECT 5
1.2. RELATED COURSES 5
1.3. OBJECTIVES 5
1.4. PROJECT PLANNING 6
1.5. EXPECTED RESULTS 7
1.6. TOOLS 7

2. ANALYSIS AND DESIGN 8

2.1. INTRODUCTION 8
2.2. WORK CONTEXT 8
2.2.1. MACHINE LEARNING 8
2.2.2. SUPER MARIO BROSS 9
2.2.3. MARIO AI FRAMEWORK 9
2.3. METHODOLOGY 10
2.3.1. DECISION TREES 11
2.3.2. INDUCTION OF DECISION TREES 12
2.3.3. INCREMENTAL DECISION TREES 15
2.4. DESIGN 18
2.4.1. AGENT 18
2.4.2. MARIO AGENT 18
2.4.3. RECORDING AGENT 19
2.4.4. PLAYING AGENT 19
2.4.5. SYSTEM REQUIREMENTS 19

3. WORK DEVELOPMENT 21

3.1. LOAD AND PROCESS DATASETS 21
3.2. DECISION TREE 22
3.3. ID3 23
3.3.1 SPLIT BY ATTRIBUTE 23
3.3.2 ENTROPY AND INFORMATION GAIN 24
3.3.3. MAKE TREE 24
3.3.4. NOISE 25
3.4. ID4 26
3.5. DEBUG VIEW 28
3.6. RECORDING AGENT 29
3.7. MARIO AGENT 30

3.8. CUSTOMIZED DATASETS 34

4. CONCLUSIONS 38

4.1. DECISION TREE LEARNING METHODOLOGIES 38

4.2. MARIO AGENT LEARNING THOUGH RECORDED DATA 38

4.3. MARIO AGENT LEARNING THROUGH CUSTOM DATA 38

5. FUTURE WORK 39

A. REFERENCES 40

B. PROJECT ACCESS 41

LIST OF FIGURES

Figure 1: Infinite Mario Bros. 9
Figure 2. Decision Tree example 11
Figure 3: Decision tree class diagram 12
Figure 4: ID3 class diagram 14
Figure 5: Decision tree with ID4 format 15
Figure 6: Decision tree ID4 example with the initial set 16
Figure 7: Decision tree ID4 example first increment 17
Figure 8: Decision tree ID4 example second increment 17
Figure 9: The agent-based AI model 18
Figure 10: Debug canvas view. 28
Figure 11: Debug view with obstacles. 29
Figure 12: Debug view with Mario representation. 29
Figure 13: Debug view with observation areas. 29
Figure 14: Mario killed by an enemy at the early stages of the training. 31
Figure 15: Mario blocked by a pipeline. 31
Figure 16: Algorithms comparison graph. 36

1. TECHNICAL PROPOSAL

This chapter presents an overview of the Final Degree Work project. The following sections include the motivation of the project, a list of the principal objectives to achieve, the planning of the development process, and its expected results.

1.1. INTRODUCTION AND MOTIVATION OF THE PROJECT

The project aims to compare the performance of online and offline induction of decision trees algorithms. This comparison is done by implementing agents of Induction of Decision Tree algorithm(ID3) and Incremental Decision Tree algorithm(ID4) in the Mario AI Framework.
The Mario AI framework is a framework for using AI methods with a version of Super Mario Bros.

The first part of the project will be focused on implementing the data structures of the decision trees that are used by the learning algorithms, in order to improve the understanding of the decision process and the later tree build process. The next step will be the implementation of the learning algorithms. These algorithms will be tested with simple example sets extracted from examples provided by the analyzed literature.

The next part will be to use the previously implemented algorithms to train an agent in the Mario AI Framework that will learn to play the classic platform game. The training process will be carried out by providing sets of examples to teach simple behaviors to the playing agent. Then a recorder agent will observe other playing agents, which are already implemented in the version 0.8.0 of the framework, in order to collect a set of examples from expert players.

The motivation underlying this project is applying my previous knowledge about the artificial intelligence field and improving them by the analysis and implementation of two learning algorithms.

1.2. RELATED COURSES

- VJ1215 Algorithms and data structures.
- VJ1224 Software Engineering.
- VJ1227 Game Engines.
- VJ1231 Artificial Intelligence.

1.3. OBJECTIVES

- Implementation of decision trees data structure and algorithm.
- Provide a visual representation of decision trees.
- Implementation of the ID3 algorithm.
- Implementation of the ID4 algorithm.
- Understanding and integration of Mario AI Framework base code.
- Integration of ID3 and ID4 learning agent.
- Provide a study and comparison of learning algorithms applied to platform games.

1.4. PROJECT PLANNING

Submission	Due by
Technical proposal	16-dic.-2019
Analysis and Design Doc.	10-feb.-2020
Mentor's report	8-jun.-2020
Report's First Draft	3-jul.-2020

Tasks	Estimated Completion Time (hours)
TECHNICAL PROPOSAL	**10**
Research	4
Write the proposal document	6
ANALYSIS AND DESIGN DOCUMENT	**86**
ANALYSIS	50
Documentation Decision Trees	10
Documentation ID3	20
Documentation ID4	10
Documentation Mario AI	10
DESIGN	36
Decision Tree	6
ID3 algorithm	16
ID4 algorithm	10
Comparison criteria	4
IMPLEMENTATION	**120**
Datasets	4
Create and read CSV files	2
Datasets as JSON	2
Decision Trees	16
Data structure	8
Decision functions	8
ID3	60
Data structure	12
Algorithm	16
Agent	10

	Learning graphic representation	6
	Learning test	6
	Results Analysis	10
ID4		40
	Algorithm	14
	Agent	10
	Learning test sessions	6
	Results Analysis	10
FINAL REPORT		**70**
PROJECT DEFENSE PREPARATION		**14**

TOTAL COMPLETION TIME	**300**

1.5. EXPECTED RESULTS

With the work developed for this project, I expect to obtain two working implementations of the ID3 and ID4 algorithms and be able to apply them to platform games in order to train an artificial intelligence agent to play them.

1.6. TOOLS

- Mario AI Framework 0.8.0
- IntelliJ IDEA Community 2020.1 (JDK 13.0.2)
- Git 2.24.1.windows.2
- Gson 2.8.6
- Google Sheets.
- Lucidchart.

2. ANALYSIS AND DESIGN

This chapter offers a general overview of the work within the context of machine learning and the Mario AI Framework. It also provides a detailed description of the machine learning techniques, providing information about their knowledge representation model, and a brief description of the algorithms.

2.1. INTRODUCTION

The work of the project lays on the development of machine learning agents for the classic platformer video game Super Mario Bros. The learning algorithms used by the agents are Induction Decision Trees and Incremental Decision Trees.

The two learning methods are implemented in order to test the efficiency of both when approaching the problem of teaching an agent to play this genre of video games.

2.2. WORK CONTEXT

2.2.1. MACHINE LEARNING

Machine learning is considered one of the central research areas in artificial intelligence. The field of machine learning is focused on how to construct systems that automatically improve with experience.

Quinlan defends that [4]: "*any attempt to understand intelligence as a phenomenon must include an understanding of learning*."

This concept of learning is defined by Herbert Simon as [5]: "*any change in a system that allows it to perform better the second time on the repetition of the same task or another task drawn for the same population*."

Simon's definition describes learning as a process of generalization from a given experience. During this process, the performance of the learning system should improve by repeating the same task and also a similar task in the domain. Since the learning domains are usually large, the learner tends to perform an induction task of the knowledge. This means that only some of all the possible examples are examined by the learner. It is the learner who, from this limited experience, must generalize correctly to unseen instances of the domain.

The use of learning techniques applied in games has the potential to provide a consistent challenge for players but also to reduce the effort to create game-specific AI [6].

Learning methods can be classified into several groups depending on different aspects such as, what is being learned, the effects of the learning on a character's behaviour, and when the learning occurs. This last point shows the biggest difference between the two techniques of this project. The learning of the Induction of Decision Trees algorithm is done offline when the game is not being played. On the other hand, the Incremental Decision Trees algorithm is able to modify its knowledge online, while playing the game.

It is important to balance the effort when working with learning algorithms. To balance the effort means that the work of programming and training the learning algorithm is less than

hardcoding the particular AI behaviours . The development of this project also aims to test whether the work is worth the effort for the results obtained.

2.2.2. SUPER MARIO BROSS

The game used to teach the learning algorithms is Super Mario Bros., a 2D side-scrolling action game. The player takes control of the famous plumber Mario. The mission of the character is to find and rescue the Princess through the Mushroom Kingdom. On his adventure, Mario will climb mountains, fight turtle soldiers, jump pits and avoid different traps.

The player can walk and run to the right and left, jump and shoot fireballs (when Mario is at Fire state). Mario can be in three states: Small (initial state), Big (there are some objects that he can crush by jumping from below), and Fire (can shoot fireballs).

The main goal of the level is to get to the end of the level. To do so, the player has to traverse it from left to right. There are some alternative goals for the levels: complete it as fast as possible, collect all the coins in the level, or getting the highest score (collect coins and kill enemies).

In the game, the control pad is simple: one d-pad and two action buttons. The player uses the D-pad to move Mario, button A to jump and B to run/shoot. Keeping in mind that the D-pad has four directions but the up direction is not used, the controls are limited to five buttons with two possible states: pressed or released. This gives the player 32(2^5) different actions. However there are some combinations of buttons that are pointless and not commonly used, like pressing right and left simultaneously.

2.2.3. MARIO AI FRAMEWORK

The environment where the algorithms are trained and tested is the Mario AI Framework 0.8.0[1], a benchmarking software that can be interfaced with learning algorithms and which origin and purpose is described elsewhere [7].

Figure 1: Infinite Mario Bros.

[1] Mario-AI-Framework git repository: https://github.com/amidos2006/Mario-AI-Framework

Many papers have been written about this benchmarking software providing the tool with significant documentation. The framework started as a modified version of Markus Persson's Infinite Mario[2] a public domain clone of the classic Super Mario Bros (Figure 1).

The framework provides a wide variety of levels including the first fifteen original levels. It also allows the user to generate new levels that can be played either by a player of an AI agent. With the source code of the framework, they are included also some of the agents that have been implemented using this tool. There is also the possibility to run the game with player controls and an agent that will take random decisions.

The application programming interface (API) of the Mario AI Framework provides the option of removing the realtime elements of the game so that the learning algorithm can step forward the game without dependency on graphical output. The API is broken down into the following Java interfaces [8]:

1. Environment interface: It describes the game state to the agent at each time step. The information is presented as:
    - Receptive area observations, as two-dimensional arrays that describe the world around Mario with block resolution. At the centre is Mario itself.
    - Exact positions of enemies with pixel resolution for more detailed information.
    - Mario state (Small, Big, Fire). There are also other binary/discrete variables to check whether Mario is on the ground, can jump, and is carrying a Koopa's shell.
2. Agent interface: It needs to be implemented in order to create a functional playing agent. The core method is the *getAction* which takes as input the environment and returns a five-bit array specifying an action.

This benchmark was originally created as the environment of The 2009 Mario AI Competition [9]. Initially, the competition only hosted a gameplay track, where the goal was to clear as many levels as possible. The competition kept growing over the years and in its second edition, it consisted of four separate tracks: *gameplay*, *learning*, *Turing test,* and *level generation*.

This work is going to focus on the *Learning Track*. Here the agents were tested on unseen levels during the agent development, but they were allowed to train on the track before being scored. Each agent was allowed to play each testing track 10000 times and it was scored only from the 10001st playthrough.

Moreover, the learning techniques of this project are mostly the one from the other on the data collecting process and the performance of the induction or increment of the decision rule. Therefore the training process will be also taken into account for the comparison of results.

2.3. METHODOLOGY

This chapter offers a detailed description of the machine learning techniques, providing information about their knowledge representation model, and a brief description of the algorithms.

[2] Infinite Mario: https://openhtml5games.github.io/games-mirror/dist/mariohtml5/main.html

2.3.1. DECISION TREES

Both learning methodologies are members of the TDIDT family. As members of this family, they represent acquired knowledge as decision trees. The construction of the trees begins with a root that proceeds down to the leaves of the tree. This is why the name *T*op *D*own *I*nduction of *Decision T*rees is given to the group of methodologies.

Decision trees are based on the idea of a universe of objects, described using attributes. These attributes measure important aspects of the object, and they take a set of discrete and exclusive values. The objects of the universe are classified into one of a set of mutually exclusive classes.

The leaves of the tree contain the classes of the universe. The other nodes of the tree represent tests based on the attributes of the object, with a branch for each possible outcome. The objects are classified starting from the root of the tree and moving down to the leaves following the branches with the values of the object's attributes.

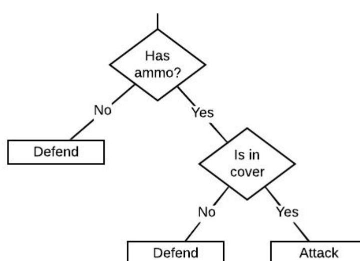

Figure 2. Decision Tree example

Figure 2 shows a representation of a decision tree from the dataset proposed by Millington and Funge at [6]. Some of the datasets from this chapter are used in the following section to illustrate other concepts. The tree has two possible outcomes: *Defend* or *Attack*. The objects of this example are represented with the attributes: *Has ammo* and *Is in cover*. In this case, the attributes only take binary values, but they could take a larger range of discrete (*i.e.* empty ammo, low ammo, and full ammo).

Decision trees take as input a tree definition that consists of decision tree nodes. The base class of these trees is **DecisionTreeNode**. It is implemented as an interface with a definition of the function *makeDecision()* which recursively walks through the tree.

There are two classes that extend from the DecisionTreeNode interface: **Action** and **MultiDecision**. For this project, the Decision nodes will be supporting multiple daughter nodes, although if the problem does not require it, they can just be implemented as nodes of a binary tree.

Decision Trees follow the structure shown in Figure 3. The private variable *root* is of type DecisionTreeNode, meaning that it can be either a **MultiDecision** or an Action. The last case, although very unlikely, is possible if the decision for any given Observation is always the same action. The public method *makeDecision()* of a Decision Tree gets an Observation and calls the *makeDecision()* method of its root, returning an action. The **Observation** class only has a public variable *attributes* which maps every attribute with its value.

As it has been pointed before, Action and MultiDecision extend from DecisionTreeNode. Since the actions represent the leaves of the tree, the *makeDecision()* method of this class

returns the Action object itself. At a MultiDecision node, the *makeDecision()* method recursively calls the function on the node returned by *getBranch()*, the daughter node mapped with the *testValue*.

This implementation is based on the one proposed by Millington and Funge in the chapter Decision Making of their book *Artificial Intelligence for Games* [10]. Here the nodes are stored in a single block of memory, avoiding slower execution and memory cache problems. The performance of the algorithm is linear with the number of nodes visited. The mentioned chapter points out that it is common for the decisions to take constant time. Assuming that the tree is balanced and the constant time, the performance of the algorithm is O($\log_2 n$), where n is the number of decision nodes of the tree. As we will see below, the trees built by ID3 tend to be balanced.

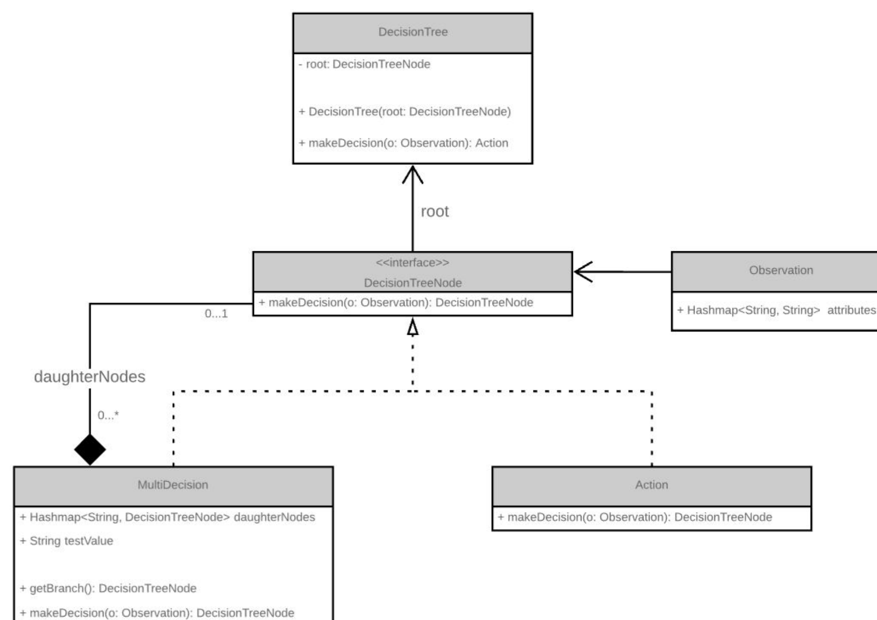

Figure 3: Decision tree class diagram

2.3.2. INDUCTION OF DECISION TREES

The learning strategy used by the systems of the TDIDT family is non-incremental learning from examples. As we will see in Incremental Decision Trees there are some variations that can be applied to these systems to make them incremental learners. The learning strategy needs a training set from which it will induct a classification rule, expressed as a decision tree.

For the induction task, the system is provided with a set of examples to be classified. The classification is carried developing a decision tree from the top down. This tree is not developed guided by the order in which the examples are provided, but by the frequency of information.

The induction of decision trees is iterative: from the set of observations, it chooses the best attribute to divide the examples into several subsets. This attribute is the one that

provides the division with the highest information gain. The algorithm repeats the subdivision process until all the examples in the subdivision are from the same class.

The algorithm takes the entropy of the classes in the set in order to choose the division attribute. It measures the degree in which the classes from the set of examples agree with the other classes of the same set. If all the examples are from the same class, then the entropy of the set is 0. On the other hand, when the distribution of the classes is uniform the entropy is 1.

The information gain is calculated as the reduction in overall entropy. The entropy is determined using Shannon's Entropy [11] function:

$$E(x) = -\sum_{i=1}^{n} p(x_i) log_b p(x_i)$$

where *n* is the number of examples and *b* the number of possible actions.

INFORMATION GAIN EXAMPLE:

The following example illustrates the calculus of the information gain for a given set [6]:

Action	Health	Cover	Ammo
Attack	Healthy	In Cover	With Ammo
Attack	Hurt	In Cover	With Ammo
Defend	Healthy	In Cover	Empty
Defend	Hurt	In Cover	Empty
Defend	Hurt	Exposed	With Ammo

This example set shows two possible actions: Attack and Defend. The entropy of the complete set is:

$$E_s = -p_{attack} \cdot log_2(p_{attack)} - p_{defend} \cdot log_2(p_{defend)}$$

$$E_s = -2/5 \cdot log_2(2/5) - 3/5 \cdot log_2(3/5)$$

$$E_s = 0.971$$

The set is divided into three attributes: *Health, Cover,* and *Ammo*. The divisions split the possible values of each attribute and the entropy for each of these values is:

$E_{healthy}$ = 1.000 E_{hurt} = 0.918

E_{cover} = 1.000 $E_{exposed}$ = 0.000

E_{ammo} = 0.981 E_{empty} = 0.000

The entropy of *Hurt* is established with:

$$E_{hurt} = -p_{attack} \cdot log_2(p_{attack)} - p_{defend} \cdot log_2(p_{defend)}$$

$$E_{hurt} = -1/3 \cdot log_2(1/3) - 2/3 \cdot log_2(2/3)$$

$$E_{hurt} = 0.918$$

As it has been mentioned before, the information gain of each division is obtained from the reduction of the overall entropy (0.971). This reduction is given by the formula:

$$G = E_s - p_{\top} * E_{\top} - p_{\perp} * E_{\perp}$$

For example, the information gain for *Health* is:

$$G = E_S - p_\top * E_\top - p_\bot * E_\bot$$
$$G = 0.971 - 2/5 * 1 - 3/5 * 0.918$$
$$G = 0.020$$

And the information gain for each of the attributes is:

$G_{healthy}$ = 0.020

G_{cover} = 0.171

G_{ammo} = 0.420

For the given set of examples the algorithm will choose *Ammo* as the best attribute for the division. It will repeat the same process for each of the divisions, but it will remove the *Ammo* attribute from the calculus. The algorithm stops when entropy is 0 so all the examples from the set converge in the same class.

The Induction of Decision Trees algorithm takes as input a list of examples, a list of attributes and the root of the tree, which is a new instance of the DecisionTreeNode described above.

Examples, from the input list of the algorithm, stores the action of the example and a hash table with its attributes. The key identifies the name of an attribute and the value of the pair its value. This class has a constructor in order to instantiate the examples from a JSON file and a *getValue()* method that returns the value of a given attribute.

The algorithm starts with the recursive *makeTree()*. This method divides the set of examples until the examples lead to the same action. The *splitByAttribute()* method carries out the division, taking the list of examples and the attribute from which divide the set, and returning a list of lists of examples. This function is called for each of the attributes of the input, and for each of these divisions, the algorithm calculates its entropy with *entropyOfSets()*. The entropy of the sets is calculated with the *entropy()* method, using Shannon's Entropy function, explained earlier in this chapter.

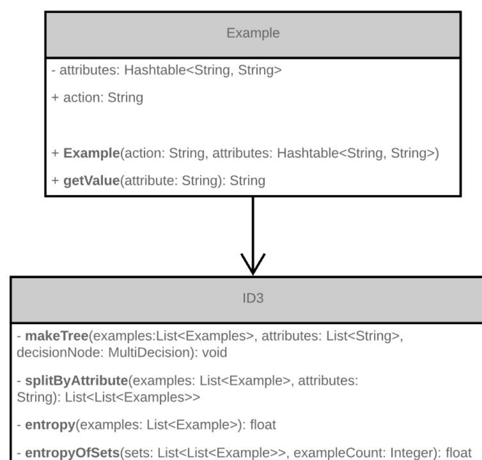

Figure 4: ID3 class diagram

At each iteration of the recursive function, the algorithm chooses the attribute with the highest information gain, subtracting the entropy of the set from the overall entropy. The input DecisionTreeNode sets its testValue as the bestSplitAttribute and maps its daughter nodes with the remaining attributes excluding the testValue. And for each daughter calls the recursive function. The stop condition of the recursive calls is when the initial entropy of the set is 0, meaning that all the examples of the set agree on the same action. The current decision node is now a leaf of the tree (an action).

The implementation of the ID3 algorithm is also based on Millington and Funge's book, in the chapter Learning [6]. Its performance is $O(a \cdot \log_v n)$ in memory and $O(a \cdot v \cdot n \cdot \log_v n)$ in time, where a is the number of attributes, v is the number of values for each attribute and n the number of examples in the initial set. The class diagram of this decision tree structure is represented in Figure 4.

2.3.3. INCREMENTAL DECISION TREES

The Incremental Decision Tree(ID4) technique goes beyond ID3 and introduces the possibility to add new examples to a classification rule that has been already inducted from a previously given set of examples.

The incremental algorithm starts as the basic induction one, generating a decision tree from an initial set of examples. In this case, the nodes of the tree keep track of all the examples classified through the branch. Every time that *makeTree* creates a new node, it adds the input list of examples from the current iterative call. The format of the decision tree with the list of examples for each node is represented by Figure 5.

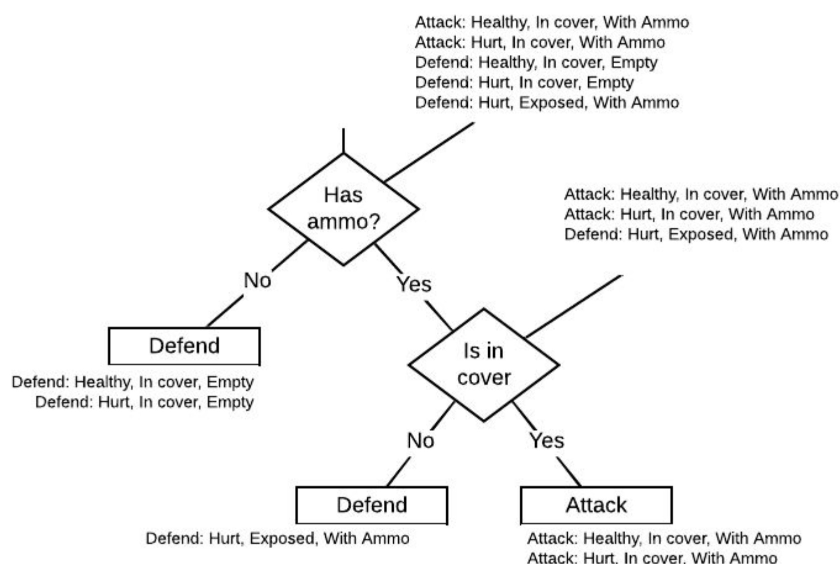

Figure 5: Decision tree with ID4 format

In addition, the nodes can be asked to update their subtree when a new example is added. There are four ways in which a node can be updated:

- If it is a leaf node and the new example belongs to the same class, this example is added to the node's object list.

- If it is a leaf node but the new example does not belong to the same class, the leaf node is turned into a decision node and it has to be determined by the division attribute. This division is like the one in the ID3 algorithm.
- If it is a decision node it has to proceed as the case below. At this point there are two possible situations:
  - The new best division attribute is the same it was before adding the new example. In this case, the classification continues from the child nodes.
  - The new best division attribute is different from the one in the original tree. In this case, the subtree that has the current decision node as root is destroyed and rebuilt (the same way we build a tree with ID3) with all the examples that the subtree was classifying plus the new example.

The node update is carried out within the function *incrementTree*. It takes as input the example that is going to be added to the tree and a node to which add the new example. The algorithm checks if the node has to be updated. The algorithm starts with the root node of the tree and recursively classifies the example until it reaches a leaf or the input node requires to be updated. The methods used to find the best split attribute are the same described for the ID3 algorithm.

INCREMENTAL DECISION TREE WALKTHROUGH:

The following example illustrates the increment process of an ID4 decision tree for a given set (Millington and Funge, 2006):

Action	Health	Cover	Ammo
Run	Healthy	Exposed	Empty
Attack	Healthy	In Cover	With Ammo
Attack	Hurt	In Cover	With Ammo
Defend	Healthy	In Cover	Empty
Defend	Hurt	In Cover	Empty

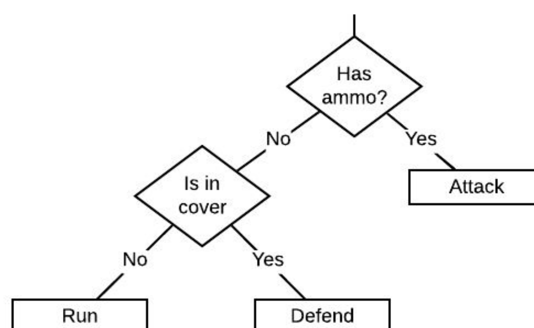

Figure 6: Decision tree ID4 example with the initial set

The initial set of examples will result in the induction of a decision tree that looks like the one shown in Figure 6. To the initial decision tree we add the following examples one at a time:

Action	Health	Cover	Ammo
Defend	Hurt	Exposed	With Ammo
Run	Healthy	Exposed	With Ammo

When the first example enters at the first decision node, the algorithm determines that Ammo is still the best attribute to use as the decision test value. The increment continues on the appropriate daughter node. This node is an action but it does not match with the example, so the action node is turned into a decision using Cover as test value. The resultant tree is shown in Figure 7.

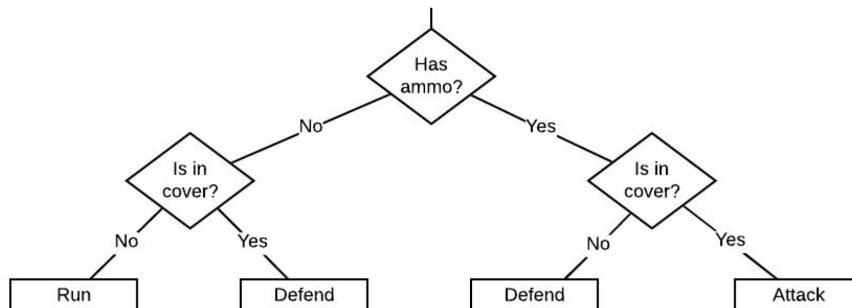

Figure 7: Decision tree ID4 example first increment

Now the second example is added to the incremented tree. At the first node, the algorithm determines that Cover is the best attribute to divide the set. In Figure 8, it can be seen that in this case, the whole tree is rebuilt.

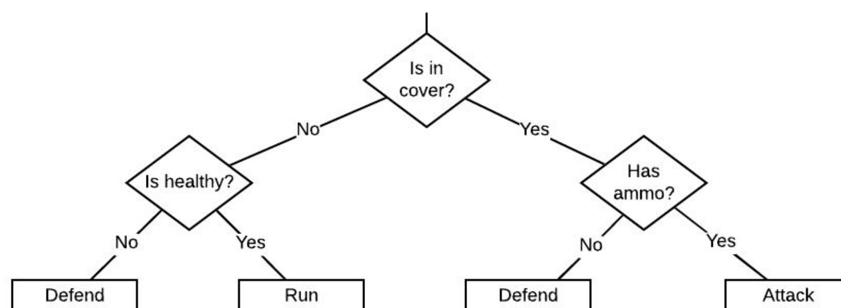

Figure 8: Decision tree ID4 example second increment

One of the main problems of the incremental algorithm is that in the worst case, every time that a new example is added to a previously built decision tree, the tree has to be built from scratch. This could happen if adding a new example meant to find a new best division attribute at the root node, which would cost more than building a new ID3 from scratch each time. However, this case seems to be very unlikely [6].

2.4. DESIGN

The following sections describe the structure and features of the agents that will collect the data through game observations and the one that will use the processed data to make the decisions when playing the game.

2.4.1. AGENT

The term “agent” will be used many times in this document, so the following section aims to provide the reader with a description of the term. In the context of the project, an agent is referred as an autonomous character that determines what action to take based on information received from the game data, and carries out those actions [12].

From this definition of agent it can be provided with a representation of an “agent-based AI model” which is represented by Figure 9

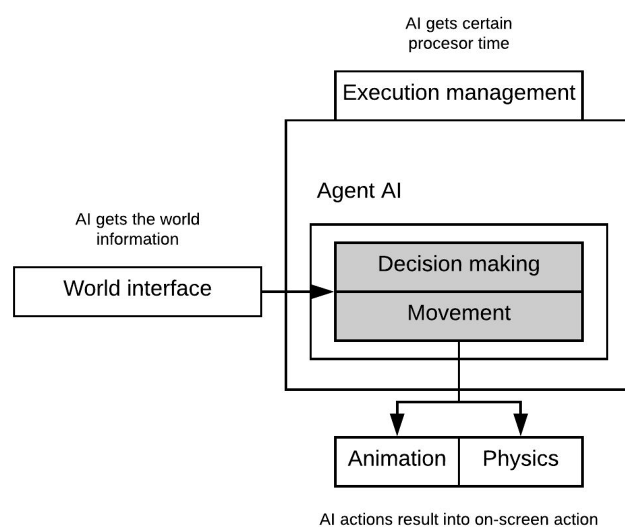

Figure 9: The agent-based AI model

2.4.2. MARIO AGENT

All the agents developed for the Mario AI Framework have to implement the interface **MarioAgent**. The interface defines the following methods:

- void *initialize*(MarioForwardModel model, MarioTimer timer):
- boolean[] *getActions*(MarioForwardModel model, MarioTimer timer): returns the action that Mario is carrying in the current game frame. It takes as input an object of the forward model for the agent to simulate the future, and the time before the agent has to return.
- String *getAgentName*(): returns the name of the agent displayed for debug purposes.

For this project there are going to be developed two different agents. One that will collect data from other agents while these play the game. The other will process the data collected by the previously mentioned agent and will make a decision tree out of it, which will be used to make decisions while playing the game.

2.4.3. RECORDING AGENT

The recording agent has its own constructor where it takes another agent instance (from now on the **recorded agent**) as input. It returns the actions based on the recorded agent's *getActions* implementation.

On the *initialize* method, the agent also initializes the recorded agent and a list of examples. At *getAction*, the agent adds this action to the example list with the action returned by the recorded agent and an observation of the current state of the world, then it returns the stored action. The observations of the world are saved as dictionaries where the name of an attribute is the key and the value is a primitive.

The recorded attributes are collected with *getObservation* which takes a forward model as input and returns the attributes Hashtable. The following table shows the name of the attributes and the methods of the forward model that return the value of the attributes:

MarioMode	*getMarioMode*
OnGround	*isMarioOnGround*
MayJump	*mayMarioJump*
EnemiesObservation	*getMarioEnemiesObservation*
SceneObservation	*getMarioSceneObservation*

When the game is over, *saveExamples* creates a new database and serializes it into a JSON file.

2.4.4. PLAYING AGENT

Due to the fact that the decision-making process is identical for the two learning algorithms, both are implemented into the same Mario Agent.

At the constructor, the agent takes as input the name of the file containing the decision tree that the agent is using as a decision-maker.

The agent implements the *getObservation* function described above and calls it at *getActions* to send it as the input of *makeDecision* which gets a node of the tree based on the observation. The action returned *getActions* is the one from the returned node of *makeDecision*.

2.4.5. SYSTEM REQUIREMENTS

Functional (F)	Non-functional (NF)
F1. The agent should be able to check the current state of the world.	**NF1.** The observations of the world is represented with debug options.

F2. The recorder agent is initialized with other agent from which record the actions.	**NF2.** The actions taken by the agents are displayed on the game screen.
F3. The recorder stores in a dataset the returned actions with an observation of the world.	**NF3.** The decision tree is represented visually on the screen.
F4. The recorder should be able to save the collected data into a dataset file.	**NF4.** The decision making process is displayed on the decision tree representation.
F5. The player agent is initialized with a dataset file.	
F6. The player agent should build a decision tree from a dataset.	
F7. The player agent makes a decision of which action it takes based on the observation of the world.	
F8. The player agent should be able to increment the dataset and update the decision tree.	

3. WORK DEVELOPMENT

This chapter describes the work done for the project, discussing intermediate results, faced problems and difficulties, and implementation decisions.

3.1. LOAD AND PROCESS DATASETS

Both learning algorithms use previous experience to start building their decision rules. These previous experiences are easy to store as Comma-separated values (CSV files).

The example dataset used to illustrate how to calculate entropy and information gain (2.3.2. Induction of Decision Trees) is represented at CSV format as:

Action,Health,Cover,Ammo
Attack,Healthy,In Cover,With Ammo
Attack,Hurt,In Cover,With Ammo
Defend,Healthy,In Cover,Empty
Defend,Hurt,In Cover,Empty
Defend,Hurt,Exposed,With Ammo

The first line (header) of the file contains the name of each attribute stored at the set of observations, excluding the first column which contains the actions of each example. From this point, every line represents an example.

There is a class **FileManager** with a static method *ReadCSV* that processes a given CSV file into a **Dataset** object. These objects store an array with all the examples of the dataset and another array with all the attributes.

To the **Example** class from Figure 3, it has been added a *printExample* method that prints at the example at the console with the structure:

Action -> *Attribute1*, *Attribute2*, *Attribute3*, etc.

Even though CSV is a good format to store actions and attributes as strings, the use of JSON format is key when working with object-based information. The **FileManager** also includes a *ReadJSON* method to store objects into JSON. The java library Gson[3] from Google is used to format objects into strings that can be later saved into text files. There are static methods to: create, write, and read text files.

The program **LoadDataFromCVS.java** shows the results of reading a dataset with CSV format, serializing it into a JSON and saving the JSON file. Then it deserializes it into a **Dataset** object and prints its lists of examples and attributes. This is the console output after executing the program with the dataset provided until this moment:

```
Reading dataset from CSV file

Attack:    With Ammo    In Cover    Healthy
Attack:    With Ammo    In Cover    Hurt
```

[3] Gson git repository: https://github.com/google/gson/blob/master/UserGuide.md

```
Defend:    Empty    In Cover    Healthy
Defend:    Empty    In Cover    Hurt
Defend:    With Ammo    Exposed    Hurt

Serializing dataset as json

Saving JSON at files
File already exists
Successfully wrote to the file

Deserializing JSON from files

Successfully read from the file

Reading deserialize dataset from JSON
EXAMPLES:
Attack:    In Cover    With Ammo    Healthy
Attack:    In Cover    With Ammo    Hurt
Defend:    In Cover    Empty    Healthy
Defend:    In Cover    Empty    Hurt
Defend:    Exposed    With Ammo    Hurt

ATTRIBUTES:
Health    Cover    Ammo
```

To test the same data processing with the dataset of Quinlan's Induction of decision trees paper comment line 13 and uncomment line 14 of the program.

```
13    String datasetFilename = "datasetMF";
14    // String datasetFilename = "datasetQuinlan";
```

3.2. DECISION TREE

For the implementation of the decision tree nodes both **Action** and **MultiDecision** extend from the abstract class **DecisionTreeNode**. This class defines a method *makeDecision* which takes as input a Hashtable of key string and object values and returns another **DecisionTreeNode**. At its constructor it initializes the list of examples that the ID4 algorithm will use when incrementing the tree.

The **Action** class has an object variable *action* that specifies the action that needs to be taken. This variable is an object so it can be represented by any primitive. For its *makeDecision* implementation, it returns itself (meaning the end of the recursive function).

On the other hand, **MultiDecision** has two variables: testValue and daughterNodes. The first one refers to the attribute used as the division value. The second one is a HashTable that maps the different values of the testValue attribute with the daughter nodes that lead to the classification of the examples.

Since the ID3 and ID4 trees implement the same methods with a significant difference at *makeDecision* (their implementation is discussed later), they extend from an abstract class **DecisionTree**. The diagram class of Figure 3 shows the methods defined by the class and its relation with the above described classes. In this case, *makeDecision* calls to the same method of the root node.

In order to test the implementation of the decision tree structure and its dependencies, it has been implemented **DecisionTreeBasic** class which has an empty constructor and the

tree is built by adding the daughter nodes hard-coded. This test is done at the **DecisionTreeMF.java** program which builds a decision tree like the one in Figure 2 and then finds the action for the observation: In cover, Empty Healthy; which determines as Defend.

```
+root:
  -test value: Ammo
  +daughter nodes:
        +Empty:
        -action: Defend
        +With Ammo:
        -test value: Cover
        +daughter nodes:
        +In Cover:
               -action: Attack
        +Exposed:
               -action: Defend

With attributes:
Cover: In Cover
Ammo: Empty
Health: Healthy

Action is: Defend
```

3.3. ID3

The implementation of the Induction of Decision Trees algorithm is done through small steps. First of all the methods that divide the set of examples according to the attributes of the examples. After that the calculus of the entropy of a set and the information gain of each division. And finally the construction of the tree using the previous functions.

3.3.1 SPLIT BY ATTRIBUTE

All the functionalities of the induction and increment of the decision trees are implemented as static methods of the class **InductionOfDecisionTrees**.

The *splitByAttribute* divides a list of examples into several subsets so each of the examples in a subset shares the same value for a given attribute. It returns a list of lists of examples and takes as input a list of examples, from which divide the subsets and an attribute as a string, used as a division of the set of examples.

The following console output shows the result of dividing the example dataset of the shooter game. This result is obtained by calling the *SplitByAttibute* method on the **MethodsID3** program:

```
============================
Division with Health
============================
Attack:   In Cover   With Ammo   Hurt
Defend:   In Cover   Empty   Hurt
Defend:   Exposed   With Ammo   Hurt

Attack:   In Cover   With Ammo   Healthy
Defend:   In Cover   Empty   Healthy
```

```
=============================
Division with Cover
=============================
Defend:   Exposed   With Ammo   Hurt

Attack:   In Cover   With Ammo   Healthy
Attack:   In Cover   With Ammo   Hurt
Defend:   In Cover   Empty   Healthy
Defend:   In Cover   Empty   Hurt

=============================
Division with Ammo
=============================
Attack:   In Cover   With Ammo   Healthy
Attack:   In Cover   With Ammo   Hurt
Defend:   Exposed   With Ammo   Hurt

Defend:   In Cover   Empty   Healthy
Defend:   In Cover   Empty   Hurt
```

3.3.2 ENTROPY AND INFORMATION GAIN

The calculus explained at the Information Gain Example (2.3.2. Induction of Decision Trees) are implemented as the static methods *entropy* and *entropyOfSets*. The algorithm uses *entropy* to find the initial entropy of the set. And *entropyOfSets* returns the entropy of a list of sets. The information gain of each set is the subtraction of the overall entropy of a subset from the initial entropy of the set. The calling of the *EntropyAndInformtionGain* method from **MethodsID3.java** program gives the following console output.

```
Entropy of example set: 0.9709506

Entropy of Hurt: 0.9182958
Entropy of Healthy: 1.0
Information gain of Health: 0.01997304

Entropy of Exposed: 0.0
Entropy of In Cover: 1.0
Information gain of Cover: 0.17095059

Entropy of With Ammo: 0.91829586
Entropy of Empty: 0.0
Information gain of Ammo: 0.41997308
```

These results agree on the calculus from the ones at the Information Gain Example mentioned above.

3.3.3. MAKE TREE

The tree is built with *makeTree* which takes as input a list of examples, a list of attributes, and a DecisionTreeNode from where the algorithm grows the tree. The method is recursive and for the initial call it takes as the list of examples all the examples from the dataset, all the attributes from the examples, and a new node as the root of the tree. At every recursion, the algorithm divides the set of examples and makes another recursion call with each subset, and with the attributes that have not been used as test values at the parent nodes yet. The

algorithm stops when the entropy of the given examples is zero, meaning that there are not any examples that can be divided or that all the examples agree on the same action meaning that the current node is an action node (leaf of the tree).

```
+root:
  -test value: Ammo
  +daughter nodes:
       +Empty:
       -action: Defend
       +With Ammo:
       -test value: Cover
       +daughter nodes:
       +In Cover:
              -action: Attack
       +Exposed:
              -action: Defend
```

Even though the algorithm is implemented using the DecisionTreeNode class which is able to store a list of examples, in this case the nodes do not add any example to this list.

A ID3 decision tree is built at the **MakeID3.java** program, which takes the shooter example as the input dataset. The resulting tree formatted with Gson into a JSON file is shown above and named *ID3Shooter.JSON* at the files folder. The distribution of the decision nodes is the same as the decision tree from Figure 2, proving that the algorithm can build decision trees correctly.

3.3.4. NOISE

The implementation of the algorithm includes an add-on to the basic ID3 algorithm to allow the tree construction work with noise in the dataset (e.g. inadequate attributes). This is done to avoid the situation where a collection of examples contains representations of both classes P and N, but the algorithm excludes further testing of the collection. The exclusion of further testing could be either because the attributes are inadequate or not able to distinguish among the examples of the collection, or because each attribute has been judged to be irrelevant to the classification of the examples. In this case, it is necessary to produce a leaf node containing the classification of the examples. The problem is that the examples of the collection are not all from the same class.

According to Quinlan, the best alternative to handle this situations is to assign the leaf to the most numerous class. This approach minimizes the sum of the absolute errors over objects in the collection [4].

The noise handling feature can be tested by commenting lines 18 and 19 of the **MakeID3.java** and uncomment lines 20 and 21.

```
18   String datasetFilename = "datasetMF";
19   String datasetFilename = "ID3Shooter";
20   // String datasetFilename = "datasetMF_noise";
21   // String datasetFilename = "ID3Shooter_noise";
```

In this case, "datasetMF_noise" contains a similar dataset to the one that has been seen already, but it adds to new examples to generate noise into the built of the dataset:

Action	**Health**	**Cover**	**Ammo**
Attack	Healthy	In Cover	With Ammo
Attack	Hurt	In Cover	With Ammo
Attack	*Hurt*	*Exposed*	*With Ammo*
Attack	*Hurt*	*Exposed*	*With Ammo*
Defend	Healthy	In Cover	Empty
Defend	Hurt	In Cover	Empty
Defend	*Hurt*	*Exposed*	*With Ammo*

As it can be appreciated, the examples in *italic* have the same values for all the attributes, but only two of them agree on the same action. The resulting tree will be very similar to the one from Figure 2, but in this case the action for exposed is *Attack*, since it is the most numerous one.

```
+root:
  -test value: Ammo
  +daughter nodes:
       +Empty:
       -action: Defend
       +With Ammo:
       -test value: Cover
       +daughter nodes:
       +In Cover:
              -action: Attack
       +Exposed:
              -action: Attack
```

3.4. ID4

Most of the implementation needed for the ID4 algorithm has already been done for the ID3 algorithm. The only difference with this previous implementation is that at the beginning of *makeTree*, in this case, it adds the given list of examples to de node's list of visited examples.

The major implementation, in this case, is the incrementTree function which takes as input the example that is going to be incremented to the tree and the current decision node at which the algorithm is adding the example. The algorithm updates the current node depending on whether it is a decision node and the new example provides a better division attribute (the algorithm rebuilds the subtree of this node) or if it is an action node and the new example does not agree on the action (the algorithms turns the action node into a decision that classifies both actions).

The fact that java does not support pointer arithmetic [13] as other programming languages like C or C++ turns the update of the nodes a bit challenging. Java does support references [14] but they do not act the same way references work in other languages. The main problem found here is that references can not be casted to an incompatible type, meaning that the bytes in memory can not be reinterpreted as some other object. The reason for this restriction is that Java is strongly type-safe.

The pointer problem is solved by asking *incrementTree* to return the updated given node. So every time that the algorithm increments an example into a node, the node is equal to the returned node of *incrementTree*. By doing this, the nodes are always updated with the corresponding reference of the node, if not, when the nodes were updated into a new extension of DecisionTreeNode (e.g. an Action turned into a MultiDecision) the new node will change its reference in memory and therefore the old one will not change.

In order to test the tree construction and the increment of examples, the program **MakeID4.java** creates an ID4 tree with the example dataset used to illustrate the algorithm walkthrough (2.3.3. Incremental Decision Trees). As can be seen in the above console output, the implementation of the algorithm is able to build, and increment trees with the expected results.

```
=============================
INITIAL TREE
=============================
+root:
  -test value: Ammo
  +daughter nodes:
       +Empty:
         -test value: Cover
         +daughter nodes:
         +In Cover:
              -action: Defend
         +Exposed:
              -action: Run
       +With Ammo:
         -action: Attack

=============================
FIRST INCREMENT
=============================
+root:
  -test value: Ammo
  +daughter nodes:
       +Empty:
         -test value: Cover
         +daughter nodes:
         +In Cover:
              -action: Defend
         +Exposed:
              -action: Run
       +With Ammo:
         -test value: Cover
         +daughter nodes:
         +In Cover:
              -action: Attack
         +Exposed:
              -action: Defend

=============================
SECOND INCREMENT
=============================
+root:
  -test value: Ammo
  +daughter nodes:
       +Empty:
         -test value: Cover
         +daughter nodes:
         +In Cover:
              -action: Defend
```

```
        +Exposed:
            -action: Run
    +With Ammo:
        -test value: Cover
        +daughter nodes:
        +In Cover:
            -action: Attack
        +Exposed:
        -test value: Health
        +daughter nodes:
            +Healthy:
            -action: Run
            +Hurt:
            -action: Defend
```

3.5. DEBUG VIEW

With the purpose of better understanding the observations collected by the agents either for recording or decision making purposes, the first step into the Mario AI framework has been implementing a debug view of the observation grid. The use of this grid and how it is obtained is detailed below.

First of all the debug details of the framework were enabled by setting to true the static boolean *verbose* from **MarioGame.java**.

```
43    public static final boolean verbose = true;
```

Figure 10: Debug canvas view.

The next step was to implement a *debugView()* method in **MarioRender**, which will only be invoked when the debug details are on. The method started drawing a transparent grey rectangle at the top left quadrant of the game view. This rectangle, represented in Figure 10, will be the canvas used to represent the debug view.

Once the reference canvas was ready, the next step was to add solid rectangles that will represent the obstacles as it is represented in Figure 11. And after that the debug view included a representation of Mario as a red line in the center of the canvas (Figure 12).

Finally, there were added empty rectangles to the debug view that represented the areas of observation (using different colours) and the obstacles showed were limited to these observation areas (Figure 13).

Figure 11: Debug view with obstacles.

Figure 12: Debug view with Mario representation.

Figure 13: Debug view with observation areas.

3.6. RECORDING AGENT

The recording agent is implemented following the design previously described. In order to collect the data, it records the performance of the Mario agent implemented by Robin Baumgarten at the first Mario AI competition [9]. This agent uses an implementation of the A* algorithm to find the shortest path to the end of the level. It has been chosen not only because of its efficiency but also because it was the winner of the first edition of the

competition due to its great performance. This controller was able to clear the 40 proposed levels with a time left of 4878 seconds in total.

At every game tick, the agent gets a set of observations and stores them in the examples list with the related actions of the recorded agent. When the game stops running (the game is over), the MarioGame class invokes *SaveDataset* from the recorder agent in case the playing agent is named “CarlosTelloRecorder” (the name of the described agent).

In order to improve the knowledge of the player agent, *getObservation* extended the number of attributes to record. Nevertheless, as it will be discussed later it did not end up being enough to learn how to properly play the game. The attributes recorded by the agent are:

Attribute	**Values**	**Description**
MarioMode	0,1,2	current Mario mode (small, large, fire)
OnGround	boolean	whether Mario is touching the ground
MayJump	boolean	is Mario able to jump
EnemiesObservation	2D grid	current enemies on the screen
SceneObservation	2D grid	current objects (not enemies) on the screen
MarioPosition	2D vector	tile location of Mario with respect to the screen
CanJumpHigher	boolean	whether Mario can press jump to reach higher positions

The name of the datasets is taken as input of the agent initialization.

3.7. MARIO AGENT

Once the recorder agent is implemented it is the turn for an agent that is able to process the data collected and develop a learning rule which it can use in order to make decisions based on different observations.

The player agent, named “LegacyAgent” is located with the recorder agent at the package agents.carlostTello. The constructor of this agent takes as input the name of the dataset from which it will induct the learning rule.

At the *initialize* method the agents initialize an array of booleans which will only change its values in case the decisionTree returns a valid action for a given observation. This is done so the agent always remembers its last action in case the decision tree does not provide another valid action. After this, the agent reads the loaded dataset and builds a decision tree with the set of examples. The resultant tree is saved as a class variable, so *makeDecision* can be invoked from *getActions*. When *getActions* gets the resultant node of the decision it checks if the node is not null and it is an action, in which case sets *action* (class variable) as the returned action of the tree. In case the returned node is null or it is not an action, *action* will remain the same as the result of the previous frame.

In order to get the input observation required for *makeDecision*, the agent calls the method *getObservation*, identical from the one implemented for the recorder agent. Two more methods: *matrixToArrayList* and *arrayToArrayList* are implemented in order to save the

vectors and grid observations as ArrayList. This is needed because when Gson formats the arrays into for the JSON, these attributes are converted into ArrayList, and the type of value from the player observation has to match the type of value saved in the tree.

Figure 14: Mario killed by an enemy at the early stages of the training.

As it has been pointed before, the set of attributes recorded on the observations did not result enough to generate a valid learning rule that allowed the player agent progress into the game. After testing different sets of attributes for the observations, the final set is the one that has provided the best results. During the first games, the player learned to start moving forward, but this was the only behavior that is supported. This behavior made Mario die every time on the first enemy or gap of the level (Figure 14). After many adjustments on the observations, Mario has finally learned to jump over an enemy but it is still not able to jump fixed obstacles (Figure 15).

Figure 15: Mario blocked by a pipeline.

Although a solution has not been found for this learning limitation the most likely reason is that the learning rule results highly specific. The main reason for this hypothesis is extracted from studying the inducted decision tree. The test value of the root node is SceneObservation. The values of the attribute are 16 by 16 grids representing the current

view of the game divided into 256 tiles. This division results in an unbalanced tree where most of the daughters of the root node are Actions.

In order to solve this problem it was tried to divide the observations of the game view into smaller pieces of the screen or use a smaller grid closer to Mario and even move this small grid forward so the focus is set on what is in front of the player. Nevertheless, it did not result in a valid solution since the above mentioned problems with the enemy and the pipeline kept occurring.

However, the simplification of the data collected pointed to an important problem. Apparently the execution of the agents, used by the recorder to collect the data, took such an irregular time to return the actions for the given observation of the world, that the data saved in the datasets ended up being distorted and corrupted. One example of this problem was noticed when the observation for an enemy standing three tiles away from Mario, was saved as if there were three consecutive enemies in front of Mario. After trying different adjustments in the data collection process, such as limiting the number of frames between collections of data, it could not be found a valid solution for the problem. For this reason, the focus was changed to provide the playing agent with simple and customized datasets in order to check if it was able to learn the basic behaviours of the character.

3.8. CUSTOMIZED DATASETS

The first behaviour that was taught to the AI agent was to jump a short distance from the ground. For the small jump, the only attribute used for the dataset was *MayJump*, which returns a boolean value that represents whether Mario is able to jump from the ground meaning that Mario is on the ground and the jump button is not being pressed. The representation of the actions, as it has been mentioned before, is by an array of booleans where each position of the array represents a different button. The order of these buttons is: *Left*, *Right*, *Down, Run* and *Jump*.

The dataset for this behaviour (MarioDatasetSmallJump) results in the following two examples:

Action	MayJump
false, false, false, false, true	true
false, false, false, false, false	false

The decision tree obtained from this data set is trivial and as it can be observed in the following representation:

```
+root:
  -test value: MayJump
  +daughter nodes:
       +true:
           -action: [false, false, false, false, true]
       +false:
           -action: [false, false, false, false, false]
```

As an extension of this behaviour, a new attribute was added to the dataset to make Mario jump as high as possible. To do so the dataset MarioDatasetHighJump adds the

attribute *CanJumpHigher*, which returns true when Mario is not on the ground but it can still jump higher by keeping the jump button pressed. The dataset also adds an example for this last situation:

Action	**MayJump**	**CanJumpHigher**
false, false, false, false, true	true	false
false, false, false, false, true	false	true
false, false, false, false, false	false	false

In this case, although quite simple, the resulting ID3 of the dataset provides a deeper understanding of the tree construction algorithm.

```
+root:
  -test value: MayJump
  +daughter nodes:
       +true:
         -action: [false, false, false, false, true]
       +false:
           -test value: CanJumpHigher
           +daughter nodes:
               +true:
                    -action: [false, false, false, false, true]
               +false:
                    -action: [false, false, false, false, false]
```

The next behaviour that was introduced was jumping over small obstacles such as blocks or enemies. The new attributes *ForwardMarioObservatio* was added to the dataset. This attribute stores a 4 by 3 matrix of booleans, where an obstacle is represented by 1 and the absence of objects by 0. Although initially there were written two datasets: one to learn how to jump over blocks and the other to skip enemies, both merged into the same one, due to the fact that the concept for both was the same: skip obstacles. This dataset has the following representation:

Action	**MayJump**	**CanJumpHigher**	**ForwardMarioObservation**
false,true,false,false,true	true	false	0000 0000 0001
false,true,false,false,false	false	false	0000 0000 0001
false,true,false,false,true	true	false	0000 0000 0010
false,true,false,false,false	false	false	0000 0000 0010
false,true,false,false,true	true	false	0000 0000 0100

<table>
<tr><td>false,true,false,false,false</td><td>false</td><td>false</td><td>0000
0000
0100</td></tr>
<tr><td>false,true,false,false,true</td><td>true</td><td>false</td><td>0000
0000
1000</td></tr>
<tr><td>false,true,false,false,false</td><td>false</td><td>false</td><td>0000
0000
1000</td></tr>
<tr><td>false,true,false,false,false</td><td>true</td><td>false</td><td>0000
0000
0000</td></tr>
<tr><td>false,true,false,false,true</td><td>false</td><td>true</td><td>0000
0000
0000</td></tr>
<tr><td>false,true,false,false,false</td><td>false</td><td>false</td><td>0000
0000
0000</td></tr>
</table>

```
+root:
  -test value: MayJump
  +daughter nodes:
       +true:
         -test value: ForwardMarioObservation
         +daughter nodes:
           +[[0.0, 0.0, 0.0], [0.0, 0.0, 1.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]:
              -action: [false, true, false, false, true]
           +[[0.0, 0.0, 1.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]:
              -action: [false, true, false, false, true]
           +[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 1.0], [0.0, 0.0, 0.0]]:
              -action: [false, true, false, false, true]
           +[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]:
              -action: [false, true, false, false, false]
           +[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 1.0]]:
              -action: [false, true, false, false, true]
       +false:
         -test value: CanJumpHigher
         +daughter nodes:
           +true:
              -action: [false, true, false, false, true]
           +false:
              -action: [false, true, false, false, false]
```

The last behaviour that has been introduced to the player agent is jumping over holes in the terrain. To do so a new attribute *GroundMarioObservation* has been added to the dataset, but in this case the previous *ForwardMarioObservation* attribute has been removed to keep the dataset as simple as possible. The new attribute *GroundMarioObservation* is represented as a 4 by 1 grid, where each row represents the tiles under Mario. The dataset for this behaviour is:

Action	**MayJump**	**CanJumpHigher**	**GroundMarioObservation**
false,true,false,false,false	true	false	1111

false,true,false,false,true	true	false	1100
false,true,false,false,true	true	false	1001
false,true,false,false,true	true	false	0011
false,true,false,false,true	false	true	0000
false,true,false,false,false	false	false	0000

```
+root:
  -test value: GroundMarioObservation
  +daughter nodes:
       +[[1.0], [0.0], [0.0], [1.0]]:
           -action: [false, true, false, false, true]
       +[[1.0], [1.0], [1.0], [1.0]]:
           -action: [false, true, false, false, false]
       +[[0.0], [0.0], [1.0], [1.0]]:
           -action: [false, true, false, false, true]
       +[[1.0], [1.0], [0.0], [0.0]]:
           -action: [false, true, false, false, true]
       +[[0.0], [0.0], [0.0], [0.0]]:
           -test value: CanJumpHigher
           +daughter nodes:
               +true:
                  -action: [false, true, false, false, true]
               +false:
                  -action: [false, true, false, false, false]
```

Finally all the previously trained behaviours were merged into one dataset named “MarioDatasetComplete”. This dataset contains a total of 21 examples with the attributes: *MayJump*, *CanJumpHigher*, *ForwardMarioObservation* and *GroundMarioObservation*. The resultant decision tree is:

```
+root:
  -test value: MayJump
  +daughter nodes:
       +true:
       -test value: ForwardMarioObservation
       +daughter nodes:
       +[[0.0, 0.0, 1.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]:
              -action: [false, true, false, false, true]
       +[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [1.0, 1.0, 1.0]]:
              -action: [false, true, false, true, true]
       +[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 1.0], [0.0, 0.0, 0.0]]:
              -action: [false, true, false, false, true]
       +[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]:
       -test value: GroundMarioObservation
       +daughter nodes:
              +[[1.0], [0.0], [0.0], [0.0]]:
                  -action: [false, true, false, false, true]
              +[[1.0], [0.0], [0.0], [1.0]]:
                  -action: [false, true, false, false, true]
              +[[1.0], [1.0], [1.0], [1.0]]:
                  -action: [false, true, false, false, false]
       +[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 1.0]]:
              -action: [false, true, false, false, true]
       +[[1.0, 1.0, 1.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]:
              -action: [false, true, false, true, true]
       +[[1.0, 1.0, 1.0], [1.0, 1.0, 1.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]:
```

```
                -action: [false, true, false, true, true]
        +[[0.0, 0.0, 1.0], [0.0, 0.0, 1.0], [0.0, 0.0, 1.0], [0.0, 0.0, 0.0]]:
                -action: [false, true, false, false, true]
        +[[0.0, 0.0, 1.0], [0.0, 0.0, 1.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]:
                -action: [false, true, false, false, true]
        +[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 1.0, 1.0]]:
                -action: [false, true, false, false, true]
        +[[0.0, 0.0, 0.0], [0.0, 0.0, 1.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]:
                -action: [false, true, false, false, true]
        +false:
            -test value: CanJumpHigher
            +daughter nodes:
                +true:
                -test value: ForwardMarioObservation
                +daughter nodes:
                        +[[0.0, 0.0, 1.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0,
0.0]]:
                            -action: [false, false, false, false, true]
                        +[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0,
0.0]]:
                            -action: [false, true, false, false, true]
                +false:
                        -action: [false, true, false, false, false]
```

With this dataset Mario is able to clear the 100% of the a simple training level: "./levels/carlosTello/completeSimpleTrainLevel.txt" and almost the 50% of the first original level.

This dataset has provided the best performance and a balanced tree structure, so it has been selected to compare the performance of the two learning techniques. For this test, both algorithms have built a decision tree incrementally from 1 example to 21. The ID3 algorithm has done this by making a new tree every time the dataset was increased. On the other hand the ID4 algorithm has incremented the dataset with the corresponding structural updates. The graph from Figure 16 shows the computational time in nanoseconds that it took each algorithm to increment a new example.

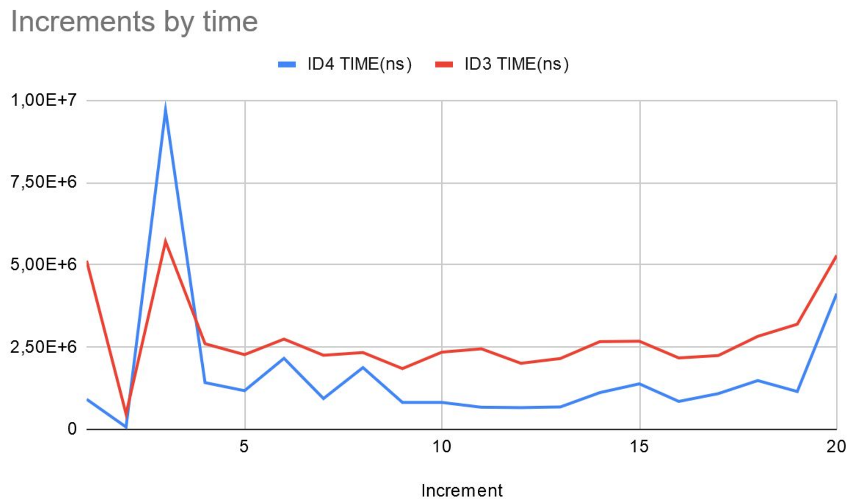

Figure 16: Algorithms comparison graph.

The values for the times of the graphs have been obtained from the program **ID3vsID4.java**.

The graph shows how the incremental process of the ID4 technique in most of the cases results in the fastest alternative to update the learning rule when the examples of the dataset are incremented.

4. CONCLUSIONS

This chapter shows and discusses the results of the development process.

4.1. DECISION TREE LEARNING METHODOLOGIES

In the first part of the project, it has been proved the efficiency of the learning methodologies from the TDIDT family to build learning rules from previously generated datasets. These techniques provide a human-friendly representation of the knowledge acquired by a machine. The mathematical representation dealing with decision trees, although not trivial, is simple enough for a quick understanding.

The main requirement of these learning algorithms is to provide them with a good set of examples that will allow the system to generalize over other unseen cases (inference mechanism). This has been the main problem of the second part of the project.

As it has been shown, through the results provided in the previous section, the implementation of the learning algorithm has proved to work as expected when the induction task is provided with well-defined sets.

4.2. MARIO AGENT LEARNING THROUGH RECORDED DATA

The results obtained in this second part of the project have not been as expected. The data sets recorded with the observations of other implemented agents, playing the game, has led to a very specific dataset unable to generalize the core behaviors of the game.

Even though the observations have been narrowed down into smaller sets of observations, the learning rule could not be properly generalized and therefore the AI agent could not perform with the expected results.

4.3. MARIO AGENT LEARNING THROUGH CUSTOM DATA

On the other hand, the results of the handwritten datasets had provided quite good results. Although these results are rather limited and can only be applied to fairly simple situations, they have proved more promising than the ones obtained through recorded data.

Nevertheless, since one of the objectives of the project was to study the reliability and performance of the algorithms ID3 and ID4 applied to platform games, from the obtained results it can be concluded that these learning techniques can be applied to this genre of games, but bearing in mind that maybe other learning techniques, as research future lines, could be tested, for the sake of better performance, over this kind of videogames.

5. FUTURE WORK

This project opens two lines of future development. On the one hand, the learning techniques that have been implemented could be tested into a different environment and situations. For example, they could be applied into other genres of games such as turn-based and/or strategy. An environment that seems more appropriate, for the algorithms of the TDIDT family, are trading card games. In these games, the action moves slower and the player is not required to make quick decisions with a constantly changing environment.

On the other hand, the project will keep growing by testing other machine learning techniques, which could be more elaborated but most likely more suitable for platform games. Some of these techniques are genetic algorithms or deep reinforcement learning, which requires much more implementation work but has proved to be very effective in many arcade games [15].

A. REFERENCES

1. "ID3 algorithm", Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/ID3_algorithm
2. "Super Mario Bros", Wikipedia the free encyclopedia. https://en.wikipedia.org/wiki/Super_Mario_Bros.
3. Mario AI Framework. http://marioai.org/
4. Quinlan, J. R. (1986). Induction of decision trees. *Machine learning*, *1*(1), 81-106.
5. Simon, H. A. (1983). Why should machines learn?. In *Machine learning* (pp. 25-37). Morgan Kaufmann.
6. Millington, I., & Funge, J. (2009). Learning. *Artificial intelligence for games*, 579-665.
7. Togelius, J., Karakovskiy, S., Koutník, J., & Schmidhuber, J. (2009, September). Super mario evolution. In *2009 ieee symposium on computational intelligence and games* (pp. 156-161). IEEE.
8. Karakovskiy, S., & Togelius, J. (2012). The mario ai benchmark and competitions. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(1), 55-67.
9. Togelius, J., Karakovskiy, S., & Baumgarten, R. (2010, July). The 2009 mario ai competition. In *IEEE Congress on Evolutionary Computation* (pp. 1-8). IEEE.
10. Millington, I., & Funge, J. (2009). Decision Making. *Artificial intelligence for games*, 293-491.
11. "Entropy (information theory)",Wikipedia,the free encyclopedia. http://en.wikipedia.org/wiki/Entropy_(information_theory)
12. Millington, I., & Funge, J. (2009). Introduction. *Artificial intelligence for games*, 3-18.
13. "Pointer (computer programming", Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Pointer_(computer_programming)
14. "Reference (computer science)" Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Reference_%28computer_science%29
15. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

B. PROJECT ACCESS

https://github.com/ctelloordonez/FinalDegreeWork.git