

Machine Learning for Smartphone Games

Dissertation presented by
Guillaume MOYSON , Quentin MEYERS

for obtaining the Master degree in
Computer Sciences

Supervisor(s)
Pierre DUPONT

Reader(s)
Pierre DUPONT, Vincent BRANDERS , Jérôme PAUL

Academic year 2015-2016

Machine Learning for Smartphone Games

Moyson Guillaume

Meyers Quentin

Submitted for the degree of Master in Computer Science

June 2016

Abstract

The objective of this thesis is to describe our attempt to create an Artificial Intelligence (AI) program capable of learning a strategy for a simple game but limited by the performance of smartphone. The first part goes through the 1st game which we had decided to work on.

Our first idea was to choose a simple game where the AI and the player had to choose among 4 actions. The goal of the player was to guess AI's move and to play it while the AI had to try not to play the same action. Unfortunately, the best strategy in this game is to play randomly. Even if we know humans can't play perfectly randomly, although we know they always have a strategy, it is almost impossible to identify it.

We then selected another game, a little more complex and for which there existed better strategies than pure chance. We will expose our conclusion about this game using different models. We'll analyse how implement various AI's that are able to learn this game and examine the performance and advantages of each of them.

Contents

Abstract	1
Introduction	1
I First game	3
1 Game overview	4
1.1 Rules	4
1.2 Data of the game	5
2 Performance testing	7
2.1 Graphs	7
2.2 Pseudo players	8
2.2.1 Random player	8
2.2.2 Preference for one move player	8
2.2.3 Preference for two moves player	8
2.2.4 Preference list player	8
2.2.5 Single pattern player	8
2.2.6 Multiple pattern player	9
2.2.7 Different-strategy player	9
2.2.8 Always different player	9
2.2.9 Mime player	9
2.2.10 Different game player	10
3 N-gram	11
3.1 Model	11
3.1.1 Smoothing	13
3.2 Results	14
4 Balanced N-grams	17
4.1 Model	17
4.2 Results	18

5	Time evaporation N-grams	20
5.1	Model	20
5.2	Results	21
6	Reinforced AI	24
6.1	Model	24
6.1.1	Java constraints	24
6.2	Results	25
7	Conclusion	27
II	A more strategic game	30
8	Game overview	31
8.1	Rules	31
8.2	Data of the game	32
8.3	Example of game	33
9	Performance testing	35
9.1	Graphs	35
9.2	Pseudo players	35
9.2.1	Random player	35
9.2.2	Positive random player	36
9.2.3	BALC player	36
9.2.4	BAPG player	36
9.2.5	SAB player	36
9.2.6	Switch strategy player	36
10	N-gram based on wins	37
10.1	Model	37
10.1.1	Choosing the move	39
10.1.2	Some words about this model	39
10.2	Results	39
11	Matrix AI	41
11.1	Model	41
11.2	Results	43
12	Opponent Tree AI	45
12.1	Model	45
12.1.1	Finding the move beating another	45
12.1.2	The Player Tree	46
12.1.3	Computing the AI probabilities	49

12.2 Results	51
12.3 With a list of best moves	52
12.4 Results with list	53
12.5 Smoothing the opponent tree AI	54
12.6 Results with smoothing	56
13 Opponent Tree AI with win	58
13.1 Model	58
13.1.1 Dual Tree	58
13.1.2 Evaluating good moves for the AI	60
13.1.3 The computation of the probabilities	60
13.1.4 Some problems with this tree	62
13.2 Results	62
14 Opponent Tree AI with time evaporation	64
14.1 Model	64
14.2 Results	64
15 Game Matcher AI	67
15.1 Model	67
15.2 Results	68
16 N-gram N-gram AI	71
16.1 Model	71
16.1.1 If we find nothing in the database	72
16.1.2 A good advantage	72
16.2 Results	72
17 Conclusion	75
 III Smartphone Application	 78
18 Application overview	79
19 Smartphone limitations	81
19.1 Database limitations	81
19.1.1 Size limitation	81
19.1.2 Access time	81
19.2 Android limitations	82
 Conclusions	 83
 Bibliography	 84

List of Figures

1.1	Picture of the game from [15]	4
3.1	Tree for N-gram	15
3.2	N-gram performance	16
3.3	N-gram performance (continuation)	16
4.1	Balanced N-gram performance	19
4.2	Balanced N-gram performance (continuation)	19
5.1	Example of tree for evaporation	21
5.2	TTL parameter evaluation	22
5.3	N-gram with time evaporation performance	23
5.4	N-gram with time evaporation performance (continuation)	23
6.1	Reinforced AI performance	25
6.2	Reinforced AI performance (continuation)	26
8.1	Round with different actions	32
8.2	Round with same action but different bets	32
8.3	Round with same action and same bet	33
10.1	Game Tree	38
10.2	N-gram based on wins AI performance	40
11.1	Matrix AI performance	44
11.2	Matrix AI performance	44
12.1	Player Tree	47
12.2	Player Tree problem	48
12.3	An example of Player Tree	50
12.4	Probability for opponent's move	50
12.5	Opponent Tree AI performance	52
12.6	Opponent Tree AI performance vs BAPG (3000 games)	52
12.7	Opponent Tree AI performance	53
12.8	Opponent Tree AI with List performance	54

12.9 Opponent Tree AI with List performance	55
12.10 Evaluation of d_c	56
12.11 Opponent Tree AI with Smoothing performance	57
12.12 Opponent Tree AI with Smoothing performance	57
13.1 Dual Tree	59
13.2 Win Opp Tree AI performance	63
14.1 Parameter evaluation of TTL for Opponent Tree AI with time evaporation	65
14.2 Opponent Tree AI with Time Evaporation	66
14.3 Opponent Tree AI with Time Evaporation	66
15.1 Database example for Game Matcher AI	69
15.2 Game Matcher performance	70
15.3 Game Matcher performance	70
16.1 N-gram N-gram performance	73
16.2 N-gram N-gram performance	74
18.1 Game interface	80

List of Tables

1.1	Example 1 for the game	5
1.2	Example 2 for the game	5
1.3	Round table in database	6
1.4	Game table in database	6
3.1	N-gram probabilities computation	12
4.1	Example of combining the probabilities	18
7.1	Players where learning is possible and impossible	27
7.2	Summary of results for first game	29
8.1	Rounds table in database	33
8.2	Example of game	34
11.1	Example of matrix	41
12.1	First history	48
12.2	Second history	49
12.3	Third history	49
12.4	Gain computation - AI moves (top) and opponent moves (left)	50
13.1	Example of computing probabilities with the Dual Tree	61
13.2	Final probabilities	62
16.1	History	71
16.2	Games following matching games	72
17.1	Summary of results for second game	77

Introduction

Nowadays many games are developed for smartphones. Many games involve two players and to allow them to be playable by a single player, developers need to create artificial players, known as Artificial Intelligence. The simplest way to implement those players is to give them a fixed strategy, or to allow them to know how good a move is based on a given situation. But if there can't be any good fixed strategy or if it is too complicated to compute the good moves, the AI will obtain poor results. The objective of this thesis is to change the approach in the creation of artificial players. Instead of implementing a fixed way of playing, we want the artificial player to be able to learn how to play the game efficiently by itself. Learning for an artificial player implies that the more it plays games, the more efficient the artificial player should become. It should be able to identify good and bad behaviours based on its experience of the game.

To reach this objective, let's first take a look at the areas related to this issue. We want to create an Artificial Intelligence program in which we will use Machine Learning techniques.

Artificial Intelligence (AI) is a field of computer sciences. The term *Artificial Intelligence* appeared in the 50's and refers to machines which received the ability to simulate human intelligence [10]. It consists in an intelligent agent that perceives its environment and interacts with it to maximise its chances of success [11]. It's a very large domain which can be divided into several specialised sub-fields [12] such as :

- Game playing : programs having the ability to play games against humans.
- Expert systems : programs that help making decisions in real-life situations.
- Natural language processing : programs that try to understand and make use of human language.
- Neural networks : systems that simulate brain operations by reproducing the types of connections into brain.
- Robotics : agents that can sense the environment and react accordingly.

Machine Learning (ML) is a kind of AI that gives the computers the ability to learn without knowing explicitly how it should perform a task [13]. A ML computer program should learn by itself how to perform a task and change its actions when new data is collected. In the context of a game, a program that it taught to learn can improve itself, i.e. its strategies should become more efficient with a view to winning. To put it simply, ML can be seen as a

way of predicting the future based on the past. We can identify three different ways to perform a machine learning task :

- Supervised learning : algorithms that learn a model from a set of labelled data and that predict label for new data.
- Unsupervised learning : unlike supervised learning, the algorithm doesn't have access to labelled data. It should first determine the labels of each data before doing the prediction.
- Reinforcement learning : an in-between algorithm that learns by rewarding the good actions and penalysing the bad ones. In that way, after some time, the agent will eventually learn the best strategy.

ML in games is useful for different tasks [9] such as :

- Learning a cost or utility function : a function that assigns a cost/utility based on the game situation, for example a minimax (see [6] for more information).
- Discovering patterns such as winning end game patterns : often based on a database. In order to do that it'll have to look for the moves that ended in a victory and extract winning patterns.
- Predicting the opponent's moves : finding the strategy that the opponent is most likely to use.

The main objective of this thesis is to develop an Artificial Intelligence based on Machine Learning techniques. It should be able to play a specific game and learn the best moves based on the games it played before. For this, we will explore different possible ML approaches that could help us devise good artificial players. Later we will develop them and optimise each of them. Each AI will have a model with different parameter and we will have to discover the most suited values for these. Finally, we will test the way in which they perform and compare it to other models.

Part I

First game

Chapter 1

Game overview

1.1 Rules

Our first game comes from DragonBall Z : Budokai 3 [15]. It's a special move (combo) from this game, that we can see in Figure 1.1.



Figure 1.1: Picture of the game from [15]

The game is pretty simple. It opposes two players, one acting as the attacker and the other one as the defender. Four actions are available, we will call them A, B, C and D. For each round, each player selects an available action. If the attacker and the defender have the same action, the defender wins the game, otherwise the attacker wins the round, and his action becomes unavailable for the rest of the game. The attacker wins the game if he wins all the rounds. As an action is removed at each round and as there are 4 actions, a game consists of up to 3 rounds.

Here are some examples :

We represented the games under the form of tables for the sake of accuracy. The rows represent the rounds of the game. The Actions column shows the possible actions at the corresponding round. The next two columns show which action were picked respectively by the attacker and

	Actions	Attacker	Defender
Round 1	A B C D	B	D
Round 2	A C D	D	C
Round 3	A C	A	C

Table 1.1: Example 1 for the game

	Actions	Attacker	Defender
Round 1	A B C D	A	B
Round 2	B C D	C	C

Table 1.2: Example 2 for the game

the defender. In the first example (Table 1.1), we see that in the first round (where all four actions A,B,C and D were available), the attacker chose B and the defender chose D. For the next round, the attacker's action is removed and thus the available actions are reduced to A, C and D. In this game, the attacker won because the defender did not manage to block any of his actions. That's not the case in the second example (Table 1.2). The defender managed to guess the attacker's action at round 2 and thus, the game ended and the defender won.

In this game, the AI player will be the attacker while the human player will be the defender and we will try to make the AI learn the human strategy. If the player plays completely randomly, it will not be possible to learn anything. However, we know humans can't play perfectly randomly (see [2] Chap. 7.3). They always play with a strategy, and it's this strategy we want to learn. For example, a human can have a preference for a certain action. Or he can never play one action. We can even go further by saying that if the AI always plays sequence $A \rightarrow B \rightarrow C$, the human will try to adapt to it. And AI could anticipate it to counter him. It's that kind of strategy that we want to learn, only because we assume that humans cannot play perfectly randomly.

1.2 Data of the game

In this game, the data that we can analyse is the following :

- Actions : A, B, C, D
- Round : a pair of actions, one for each player. We usually represent them as a tuple (a_p, a_{AI}) or as $a_p \& a_{AI}$ (in tree representations, see later) where a_p is the player's action and a_{AI} the AI's action.
- Game : a sequence of 1 to 3 rounds.

In our application, we stored the raw data in a database, as we expected to need a lot of information to build up our models. Table 1.3 shows how the rounds are stored in the database. We also stored the games in a different table in order to be able to add specific information such

as the player's ID identifying the player that played that game. An illustration of that table can be found in Table 1.4.

Rounds			
Game ID	Round nbr	Player	AI
0	1	A	B
0	2	A	C
0	3	A	D
1	1	B	B
2	1	B	C
2	2	D	D

Table 1.3: Round table in database

Games	
Game ID	Player ID
0	0
1	0
2	1

Table 1.4: Game table in database

Chapter 2

Performance testing

Before starting to describe the models, we would like to make it clear how we will test the performance of the models. By performance, we mean a good win rate (higher than a win rate of 25%, the expected win rate of a random player).

We first took sometime to compute the expected win rate for an AI playing randomly (or an attacker playing randomly). This gives us a lower bound for the performance because we would like to obtain better results than those obtained by random guessing. As our AIs will be the attackers, the calculation is the following :

$$P[W_i] = \text{expected win rate of the attacker at round } i$$

$$P[W_1] = 3/4$$

$$P[W_2] = 2/3$$

$$P[W_3] = 1/2$$

$$P[\text{Win}] = P[W_1] * P[W_2] * P[W_3] = \frac{3}{4} * \frac{2}{3} * \frac{1}{2} = \frac{1}{4}$$

It's a multiplication since it has to win all 3 rounds to win the game. That means that if our AI has a win rate that exceeds 0.25, it is an AI that has learnt a strategy and is then better than if it had used a random strategy.

2.1 Graphs

The next chapters are dedicated to the different models that we implemented. For each model, there is a section dealing with the results. In these sections, we'll show graphs to illustrate the results we obtained. Here is an explanation of what is represented on these graphs : the X-axis represents the number of games while the Y-axis represents the AI's win rate. Two things are represented on the graphs :

- Ten green lines. Each green line represents the model playing against the player in the same conditions. Those lines are the AI's win rate based on the number of victories and games.
- A straight red line at 0.25 represents the performance of the Random AI.

We decided to draw multiple green lines in order to highlight variance. Variance is the gap between the best line and the worst line. It allows us to know more precisely how the model

evolves. If we had only drawn one line we would have been able not to say if the model generally evolved in that way, or if it is an exception. Variance helps solve this question.

2.2 Pseudo players

As we can't play thousands of games ourselves to see if an AI can significantly make the difference against a human, we decided to create a lot of potential players to test the performance of our models. We will now describe all our interesting players in this section.

Note : we suppose humans will try to play this game randomly. But we know humans can't play perfectly randomly. So to simulate this way of playing at best, we assign our players a probability to play their strategy. For example, if we give it a probability of 0.50, the player will follow its strategy in 50% of the cases. We decided to set the default probability to 0.75. If the player cannot play his strategy, it plays randomly.

2.2.1 Random player

Our first player is really simple : it plays randomly. We give it a list of possible moves, and it chooses one move from to the list randomly.

2.2.2 Preference for one move player

This player simulates a player having a preference for a move if it is available, otherwise it plays randomly.

2.2.3 Preference for two moves player

This player has two preferred moves. So, when it has to play, if both moves are available, it chooses randomly between those two moves. If only one is available, it plays it. And if none of them are available, it plays randomly.

2.2.4 Preference list player

This player has an action list sorted by preference. So, when it has to play an action, it looks in its list if the first action is available among the possible moves. If it's the case, it plays it. If not, it looks at the second preferred move, and so on. What we can notice with this player is that if its first preferred action is always available, it will always play it.

2.2.5 Single pattern player

This player has a pattern that it will try to play. For example, we can say that its pattern is :

- Playing action *A* at round 1.
- Playing action *C* at round 2.
- Playing action *D* at round 3.

It's different from the *Preference list player* (see Section 2.2.4), because for each round, it will play different moves. Unlike the *Preference list player* that can always play the same action in a game.

If it cannot play a move from its pattern because of the probability introduced earlier (see remark in Section 2.2), it will play randomly, but not the move from its pattern (if the move of its pattern is available).

2.2.6 Multiple pattern player

This player has different patterns. When it has to play, it will try to play the strategy of its first pattern. If the move is not available, it will try the move of its next pattern, and so on. If it finds no possible move in its patterns, it plays randomly.

For example, consider the following patterns :

1. $A \rightarrow B \rightarrow C$
2. $A \rightarrow C \rightarrow B$
3. $A \rightarrow D \rightarrow C$

Let's now consider that the possible moves for this player are C and D and that we are at the second round. Our player, if probabilities allow it, will play its strategy, and so will try to play its first pattern. For the second round, the move of its first pattern is B . But B is not a possible move. So, it will look into its second pattern, and will find C . As C is a possible move, it will play it.

2.2.7 Different-strategy player

This player plays in the same manner as the *Multiple pattern player* (see Section 2.2.6), except that it chooses the pattern to play randomly. And if the move of the pattern is not available, it plays randomly.

2.2.8 Always different player

This player never plays the same action in a game if possible. It keeps the moves it played in memory, and when it has to play a new action, it removes the actions played from the possible moves. Then, it plays randomly with the possible moves. If there are no more possible moves after the "pruning", it plays randomly with the initial possible moves.

2.2.9 Mime player

This player plays the moves that his opponent played in the previous game, in the same order.

For example, if the opponent played A then C and then D , this player will try to play A at the first round, C at the second round and D at the third round of the next game.

If the move is not available, it plays randomly.

2.2.10 Different game player

This player will try not to play the same moves it played in the previous game.

For example, if it played A then C and then D , it will try not to play A at the first round, not to play C at the second round and not to play D at the third round of the next game.

Chapter 3

N-gram

In simplistic words, learning consists in using the past to predict the future. In our case, it corresponds to using the previous games to predict the future games. According to several books, a suitable model to accomplish this task should be the N-gram.

3.1 Model

The N-gram model consists in predicting the next word in a sentence, given the N-1 previous words. For example, if we know the following sentences :

I love Machine Learning.

My thesis is about Machine Learning.

My machine is broken.

If we want to predict which word is most likely to come after "machine" (we then use a bi-gram, looking only one word backward). We will look at all the words following "machine" and decide which one is the most frequently used. In this example, "learning" is used twice after "machine" and "is" is used once. That means that "learning" is the most probable word to follow "machine" in the example. We are actually interested in the probabilities of these words to follow "machine". Here "learning" will be given a probability of $\frac{2}{3}$ and "is" of $\frac{1}{3}$. All the other words have a probability of 0. To compute these probabilities, we simply count the number of times the words appear. First, we see that "machine" appears 3 times in the sentences. Then we see that "machine learning" appears twice in the examples. The same goes for "machine is" : it only appears once.

More formally, we will say that "machine" is the *history* (denoted by h), that is the N-1 words that will be used to predict the next word. $C(h)$ is the count of h 's, the number of times that h appears in the sentences. $C(w, h)$ is the number of times that h is followed by a word denoted by w . The probability of a word w to follow a history h , is thus

$$\hat{P}(w|h) = \frac{C(w, h)}{C(h)} \quad (3.1.1)$$

So, in the example above, we have $C(machine) = 3$, $C(learning, machine) = 2$ and $C(is, machine) = 1$. Thus $\hat{P}(learning|machine) = \frac{2}{3}$ and $\hat{P}(is|machine) = \frac{1}{3}$.

If we go back to our game, the *history* represents the previous rounds of the current game and the prediction is the next defender's move. The prediction is based on the previous games that were played.

The way we implemented this model is by using a tree. Figure 3.1 represents this tree. The root of the tree is the situation where no move has been played yet in the current game. The branches of this tree correspond to all the possible rounds that can be generated from this situation. At the root, all the branches represent all the possible combinations available to both players, at the start of the game. For example, we have the branches A&A, A&B, A&C, A&D, B&A, B&B, and so on. The first action is always the player's action (defender's action), and the second action is always the AI's action (attacker's action). The information that is stored in the nodes shows how many times this sequence of moves has been played. The number in the root thus indicates the total number of games. For example, the number in the subtree right after playing A&C shows how many times the player played A and the AI played C in the same round. In Figure 3.1, the number in that node is 45, meaning that the combination A&C has been played 45 times (over 571 games). Notice that this tree has the property that the number in a node is equal to the sum of the numbers in the nodes of the subtree.

This tree is created at the very beginning extracting data from the games in the database, and is updated everytime a new game is played.

Once we have built this tree, we can infer probabilities from it. To compute the probability of the player to play action A, we go through all the branches of the current node that start by A (for instance A&A, A&B, A&C and/or A&D depending on the availability of the actions) and we sum the content of the nodes to count how many times A was played by the player. We then divide this count by the number in the current node of the tree. In these case $C(w, h) = C(A, h)$ and the number in the current node corresponds to $C(h)$. The *history* is represented by the position in the tree. Column 2 of Table 3.1 shows an example of these counts assuming that the current node contains number 420. In the third column, we compute the probability of the player to play the corresponding action.

Actions	Counts	Probability	Inverse	Normalised
A	180	$180/420 = 9/21$	$21/9 = 2.33$	$2.33/21.58 = 0.11$
B	120	$120/420 = 6/21$	$21/6 = 3.50$	$3.50/21.58 = 0.16$
C	40	$40/420 = 2/21$	$21/2 = 10.50$	$10.5/21.58 = 0.49$
D	80	$80/420 = 4/21$	$21/4 = 5.25$	$5.25/21.58 = 0.24$

Table 3.1: N-gram probabilities computation

The point now is to use these probabilities to allow the AI to play its next action. As it must not play the same action as the player; we could say that our AI should play the action with the lowest probability. Yet, with this solution, the model would become too easy to be countered by a human player. A better idea is to play the player's actions but with the inverse probability. That is illustrated in column 4 of Table 3.1. However, if we keep these values, we do not respect

the **consistency property**.

Consistency property is a basic principle in probability stating that the sum of all probabilities must be equal to 1 when we want to predict an event. We need to be cautious about this, otherwise we might end up with wrong probabilities.

For example, if we are playing with a dice, and we want to make a prediction about the numbers, we will compute probabilities. As we know all numbers have an equal probability of being drawn, we will assign a probability of $\frac{1}{6}$ to each number.

However, suppose we have a loaded dice and we know 6 is twice as likely to be drawn as opposed to the other numbers. We could simply assign a probability of $\frac{2}{6}$ to number 6. But, by doing so, when we sum all the probabilities, we obtain $\frac{7}{6}$. It is superior to 1, so it is inconsistent.

To stay consistent, we must normalise our probabilities. It consists in assigning each event its probability divided by the sum of probabilities. In our case 1, 2, 3, 4 and 5 have the same probability, i.e. $\frac{1}{6}$. After normalisation, the new probability is $\frac{\frac{1}{6}}{\frac{7}{6}} = \frac{1}{6} * \frac{6}{7} = \frac{1}{7}$. The probability for number 6 is $\frac{2}{6}$. After normalisation, its new probability is $\frac{\frac{2}{6}}{\frac{7}{6}} = \frac{2}{6} * \frac{6}{7} = \frac{2}{7}$.

In order to stay consistent, we must normalise the inverse probabilities by dividing each probability by the sum of all the others. The normalised probabilities will be the AI's probability to play a given action.

Let's take a look at the numbers in this table. We can observe that action C is the one that has been played the least (only 40 times over 420 games). This means that it could be interesting for our AI to play this move since it is less probable that the opponent plays it. This explains why, in normalised inverse probabilities, the action C has the highest probability of being played. Action D is the second less played action. Its probability of being played is twice the one of C, so it's normal for the normalised probability to be twice less likely. Finally, we notice that the most probable action for the player, A, is the one with the lowest probability of being played by our AI.

3.1.1 Smoothing

A situation that can happen, especially when we don't have a lot of data to build our model, is that there might be actions that have not been played in a particular situation even if they are available. This can cause trouble to our algorithm. For example, when we inverse the probabilities, we cannot divide by 0. For this reason, we need to assign some low probability to unseen events, this operation is called smoothing.

Going back to our N-gram example, we said we had assigned a zero probability to the words we'd never seen after a history in our data. But, just because we've never seen a word after a history doesn't mean it is impossible to see it. That's why we use smoothing.

Smoothing is a technique that assigns a non-zero probability to events we never met during the training. As a result it increases the number of events with a non-zero probability, hence the probability distribution becomes smoother [14].

In this game, we used the easiest way carry out smoothing, which is the *add-one* (or *Laplace*) smoothing. It consists in adding 1 to every count. Once it is done, we count the number of 1's we added, and we add the result to the denominator of the fraction when computing the probabilities. Since smoothing is only used to resolve some trouble with counting the probabilities, it doesn't have a significant impact on the performance of the algorithm.

3.2 Results

We presented the results of this model against the players presented in previous Chapter (2.2) in Figure 3.2 and 3.3.

Figure 3.2 shows how the N-gram model can learn enough information to allow the AI to beat some players. We show these graphs together because they share a common characteristic which is very important : there is an improvement in the win rate (except the last one). By doing so, this model becomes better than random (straight red line at 0.25) guessing against those players. We observe that this AI learns the strategies of the following players : *Single pattern*, *Multiple pattern*, *Preference for one move*, *Preference for two moves* and *Preference list players*.

On Figure 3.3 we notice that the N-gram cannot adapt its strategy against some players : *Random*, *Different game*, *Different strategy*, *Always different* and *Mime player*. It is not better than random guessing.

However, a good point is that its performance doesn't drop too low as compared to the red line (random). This model can either learn the players' strategy and have good results, or it cannot learn anything and then obtains more or less the same performance as playing randomly.

The major difference between these 2 categories of players is the randomness level. On the one hand we have the players who play in a more regular way. In this case, the model is able to learn the strategy and counter it efficiently. On the other hand, when we introduce more randomness in the player, the model can't recognise a particular behaviour adopted by the player in order to win. Typically with the *Different-strategy player*, we choose a pattern randomly instead of using always the same one (*Multiple pattern player*).

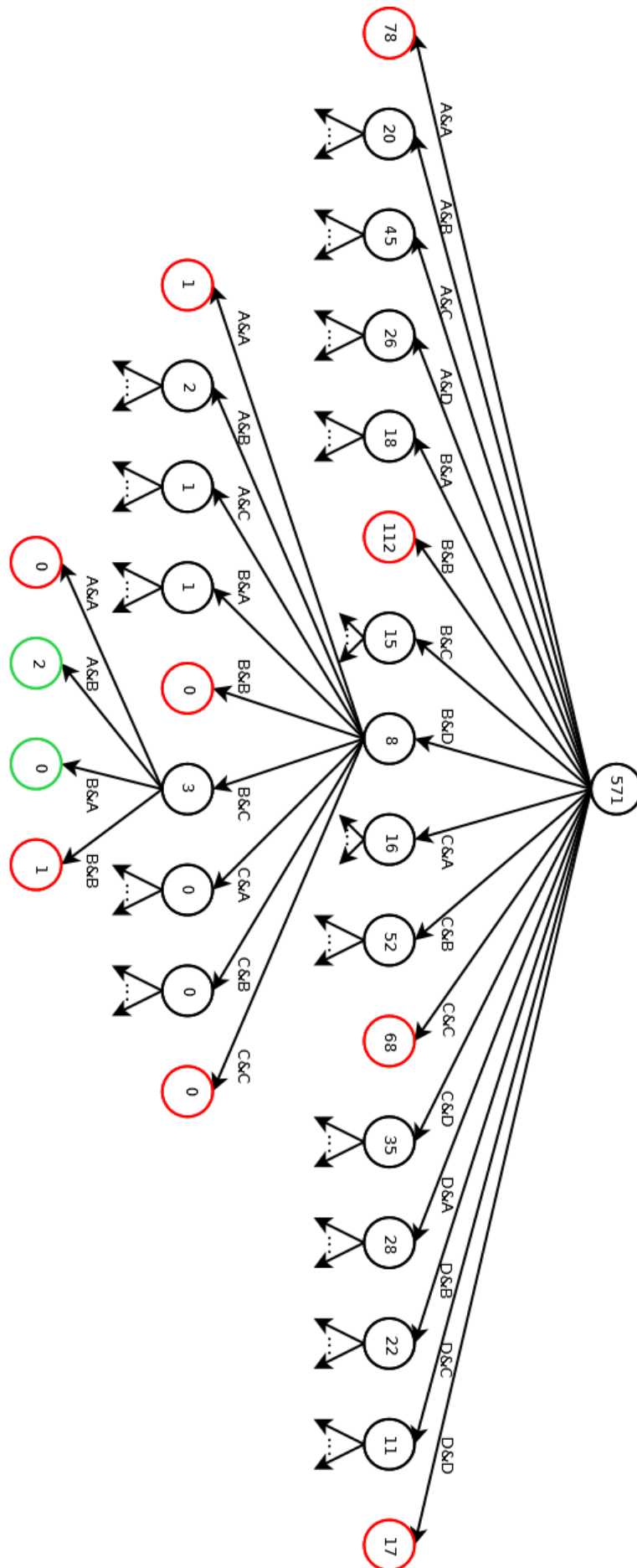


Figure 3.1: Tree for N-gram

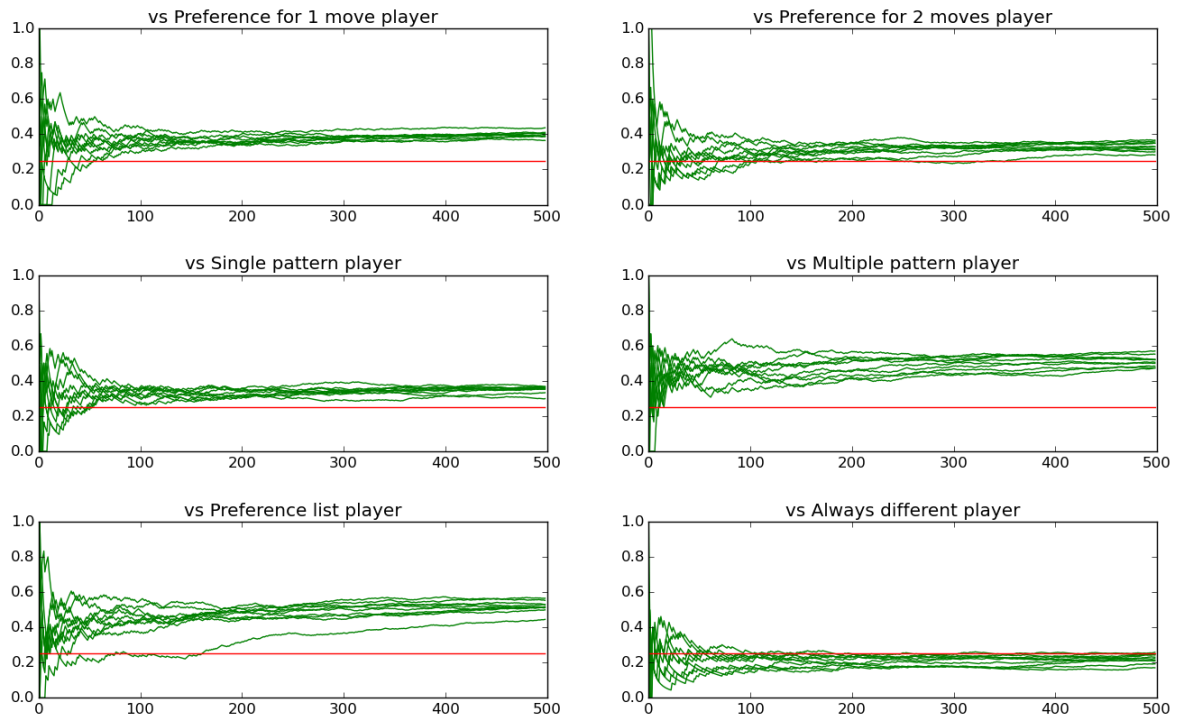


Figure 3.2: N-gram performance

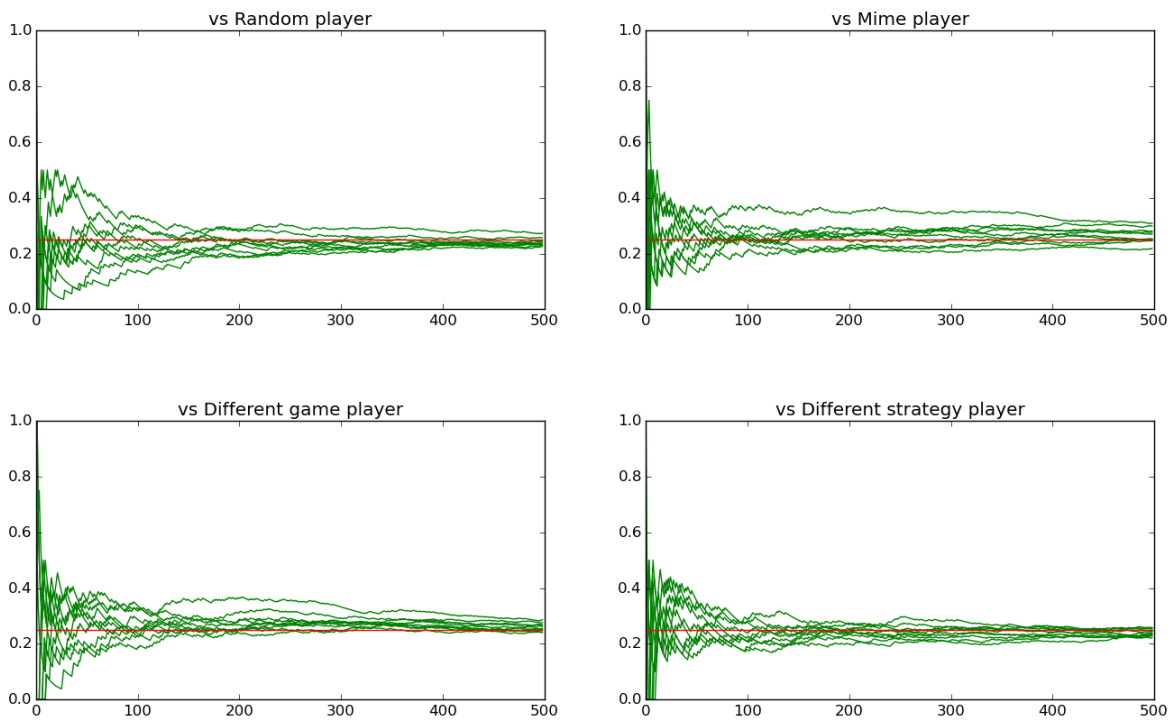


Figure 3.3: N-gram performance (continuation)

Chapter 4

Balanced N-grams

The N-gram model gave some positive results against players with a fixed strategy but we noticed that the model did not adapt quickly enough when we tested it against human players. The following model aims to improve that part of the N-gram model.

4.1 Model

With the basic N-gram model, the tree stores the information about all the games that are present in the database. But some of them may be more relevant than others. In this model, we assume that the AI plays against a new player each time the application is launched. In this case, we think that the current player's strategy is not necessarily the same as the one played by previous players (the old games in the database). That's why we add a specific tree for computing the N-gram on the games that the current player has played. This is done using the player ID in the database.

Then we compute the probabilities of the possible actions in the same way as the N-gram model but on two different sets :

1. on all data stored in the database : P_{all} .
2. on restricted data, the one corresponding to the current player : P_{curr} .

When we have no data about the current player, we want the algorithm to only take the probabilities from the all database into account. And when we have enough games from the current player, the algorithm should only take these ones into account.

Let us first show an example of what we have and what we want. Let's take the probabilities at the first round, where all actions are available. Let's assume that the full data set gives us the probabilities in the P_{all} column of Table 4.1. Let's also suppose that the N-gram on the current games of the player outputs the probabilities in the P_{curr} column of Table 4.1.

We combine the probabilities by using an α factor, that has a value between 0 and 1. We have the general formula :

$$P_{tot} = (1 - \alpha) * P_{all} + \alpha * P_{curr} \quad (4.1.1)$$

In the fourth column, P_{tot} ; $\alpha = 0.5$, we show an example of combination between the probabilities if we consider that they have the same importance. Here, the formula is :

Probabilities of playing the actions					
Actions	P_{all}	P_{curr}	$P_{tot}; \alpha = 0.5$	$P_{tot}; \alpha = 0.25$	$P_{tot}; \alpha = 0.75$
A	0.60	0.30	0.450	0.5250	0.3750
B	0.10	0.40	0.250	0.1750	0.3250
C	0.25	0.20	0.225	0.2375	0.2125
D	0.05	0.10	0.075	0.0625	0.0875

Table 4.1: Example of combining the probabilities

$$P_{tot} = (0.5 * P_{all}) + (0.5 * P_{curr}).$$

The last two columns of Table 4.1 show how the probabilities are affected when the α varies.

As we said above and this is what we want, when we have no game for the current player, P_{all} should not be affected by P_{curr} . That's why, the initial value of α is 0. Later on, the more games we have from the current player, the more P_{curr} should impact the final probabilities. After each game, we then increase α by a fixed value, between 0 and 1, that is called *step*.

The value we assign to *step* is really important because it impacts the speed at which the model will go from only considering P_{all} to only considering P_{curr} .

Let's take for example *step* = 0.10. At the first game of the new player, we only use P_{all} probabilities. After the first game, α is increased to 0.10 and thus, we will gradually take the previous games into more and more considerations into total probabilities. After five games, the value of α is now 0.5, that means that the model will consider the probabilities on the five previous games as important as the probabilities on the full data set. We see that taking a *step* of 0.10 leads to only considering the P_{curr} after 10 games. The *step* value in the example is obviously senseless and we need to find a better one. To do so, we use parameter evaluation algorithm on different basic strategies (see Section 2.2). In this algorithm, we have a loop for different values of α and within that loop, another loop for the performance estimation. That second loop performs 10 runs of the algorithm with 1000 games against a mix of different players.

These tests showed that the best value for *step* is 0.0067. That means that our model needs around 149 games of the current player to consider that these games give enough information to allow model to be able to compute good probabilities for the current player.

4.2 Results

The performance of this model is illustrated in Figure 4.1 and in Figure 4.2.

The idea of this model was to be able to learn a strategy more quickly, i.e. requiring fewer games to have a win rate above 0.25 (the red line). This only worked against the *Preference for 1 move player* : this model requires about 20 games for all the green lines to be above the red one while the basic N-gram needed about 60.

However the other results were not as good as we expected. The performance of this model is almost always worse than the previous model in the learning speed and the final win rate.

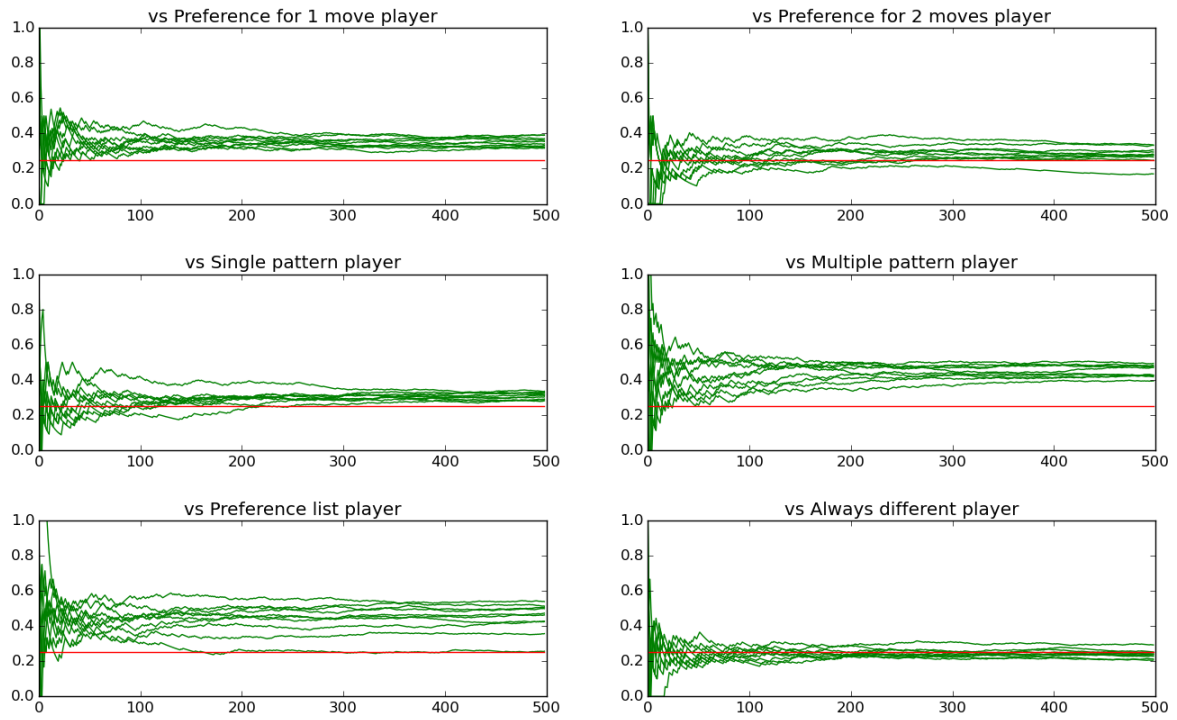


Figure 4.1: Balanced N-gram performance

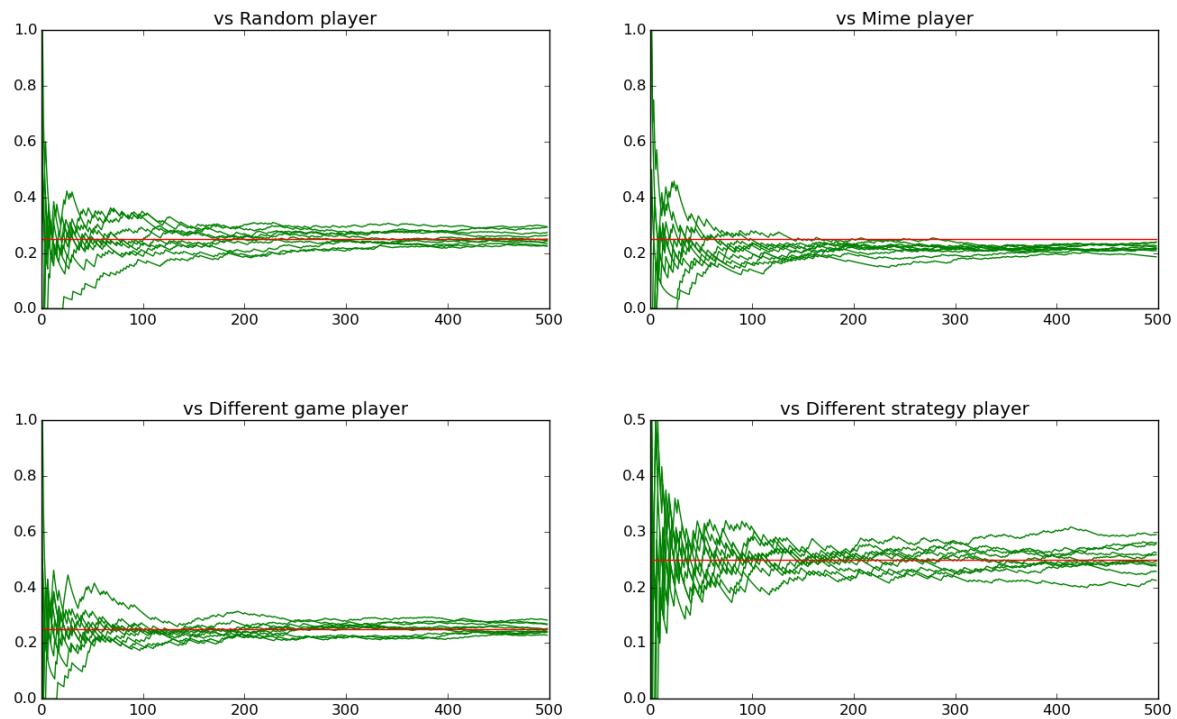


Figure 4.2: Balanced N-gram performance (continuation)

Chapter 5

Time evaporation N-grams

The problem with the previous model is that, after some time, the model limited its data to a subset and consequently become less well informed about the player's strategy. In order to keep more data into account and while keeping an eye on more recent games - the latter being more relevant than old ones - we introduced this new model. It goes further than balancing between the full data set and current player's data. The principle is simple : a game that was played a long time ago is less relevant than a newly played one. The idea behind this improvement is then to later assess the games in relation with its "age". And if the game is older than a given threshold, we will discard it (and not take it into account at all).

5.1 Model

The idea to implement this model is to slightly change the structure of the tree. Figure 5.1 represents a subtree of this tree. Instead of storing a single number inside the nodes, we will store a list of numbers. The size of this list is the total number of games that went through this node and that we consider in the model. When we have to add a new game in the tree, we will add a *special value* in the list of each node along the path corresponding to this game. We call this *special value* : TTL (Time To Live) because it represents the period of time during which this value has to be taken into account. Once the played game is added, each value inside the list of each node of the tree is decreased by one. Therefore, the TTL values that are stored in the list indicate the "age" of the game. When a value reaches 0, we remove it from the list.

Figure 5.1 represents a subtree where actions A and C are not available. In this example, we assume TTL to be 25. The fact that 25 appears in the node after the D&D branches of this tree means that the last game played reached this situation and that the player and the AI both played action D.

In this case, if we want to compute probabilities for the simple N-gram, we can just use the size of the lists. For instance, the probability for the player to play B is $5/13 = 0.385$ and for him to play D is $8/13 = 0.615$. But we want our probabilities to reflect the fact that in the last eight games (25, 24, 23, 22, 21, 20, 19, 18), the player played B 5 times, which can indicate that the player tends to have a preference for B now. The easiest way to do that is to compute the probabilities from the sum of the numbers in the list. The more recent, the higher the number,

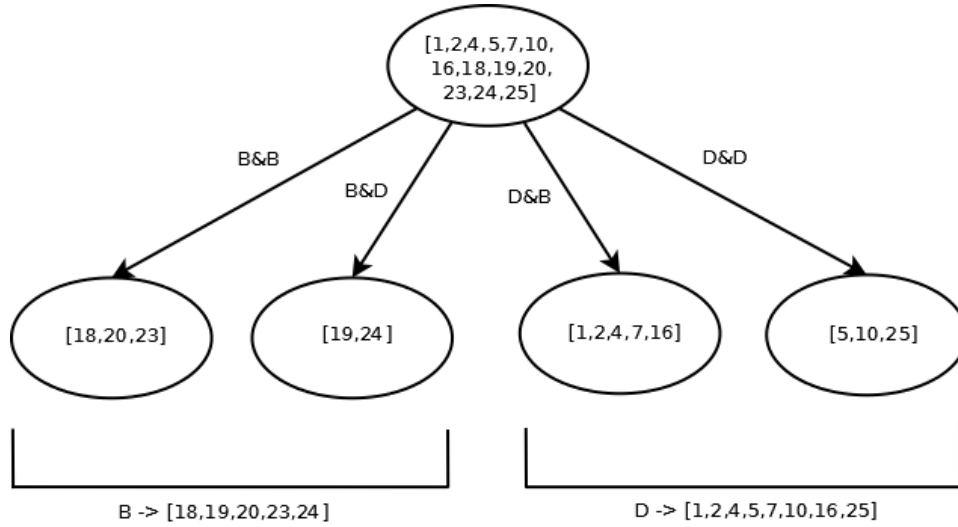


Figure 5.1: Example of tree for evaporation

the higher the probability. The formula is then :

$$P[x] = \frac{\sum SubList_x(i)}{\sum List(i)} \quad (5.1.1)$$

where SubList is the list obtained by joining the list for the same player according to action x (the lists at the bottom of Figure 5.1) and List is the list of the current node (the one on the top in Figure 5.1).

In this example, we have the sum of list :

$$\sum List(i) = (1 + 2 + 4 + 5 + 7 + 10 + 16 + 18 + 19 + 20 + 23 + 24 + 25) = 174$$

And we obtain the following probabilities :

$$P[B] = (18 + 19 + 20 + 23 + 24) / 174 = 104 / 174 = 0.60$$

$$P[D] = (1 + 2 + 4 + 5 + 7 + 10 + 16 + 25) / 174 = 70 / 174 = 0.40$$

If we compare the probabilities given by this model with the ones we'd obtained using the N-gram presented before, we see that this model realises that the player is currently playing B more often than before. Conversely the N-gram model which just considers how many times actions were played, still assumes that the player prefers D.

The important factor that will influence the model is obviously the choice of the TTL value. To find out the most suited value, we use, as in the previous chapter, a parameter evaluation technique. As shown in Figure 5.2, the best value is 585.

5.2 Results

We'll now introduce the performance of the N-gram with time evaporation against all the players. Figure 5.3 and 5.4 sum them up.

Compared with the two previous models, we observe that this model is really better. Indeed, it can beat 6 players : *Preference for 1 move*, *Preference for 2 moves*, *Single pattern*, *Multiple*

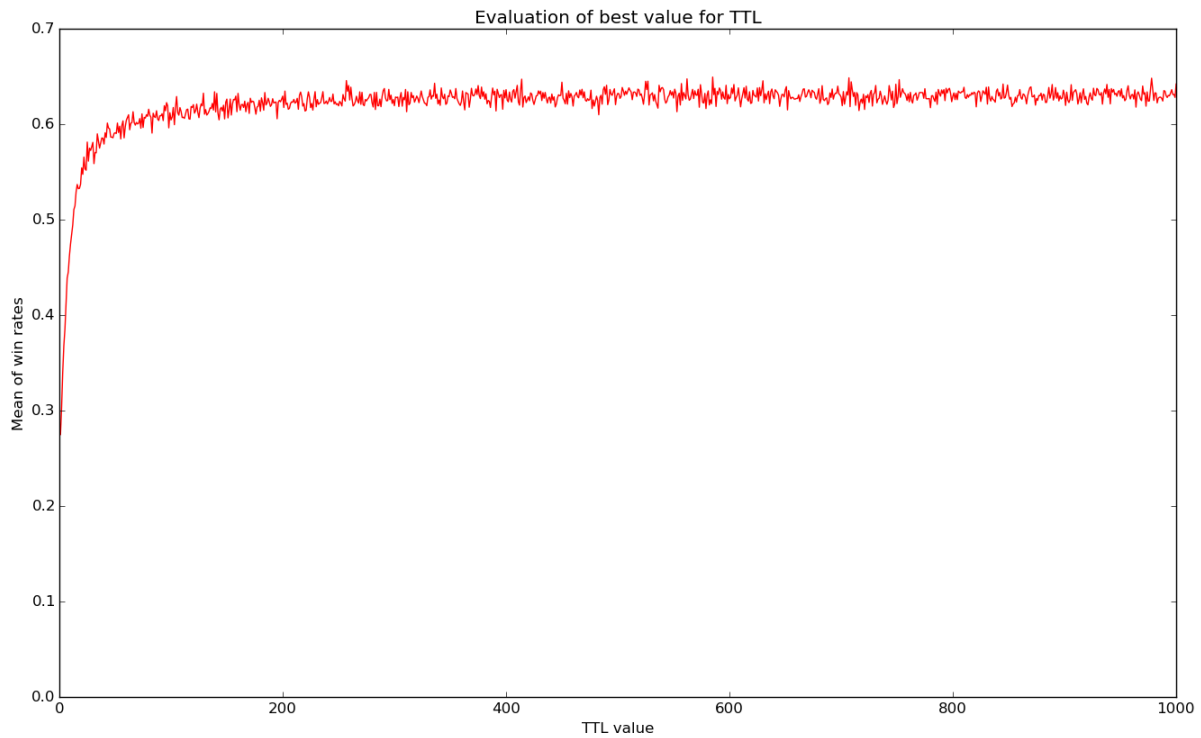


Figure 5.2: TTL parameter evaluation

pattern, *Preference list* and *Always different players*. Moreover, it can beat one model the two previous models couldn't : the *Always different player*.

Another thing we can notice is the difference of variance (the gap between the best green line and the worst green line). Indeed, the variance in this model is smaller than the one of the previous model. It means this model is more stable, and we can therefore be more confident in the results.

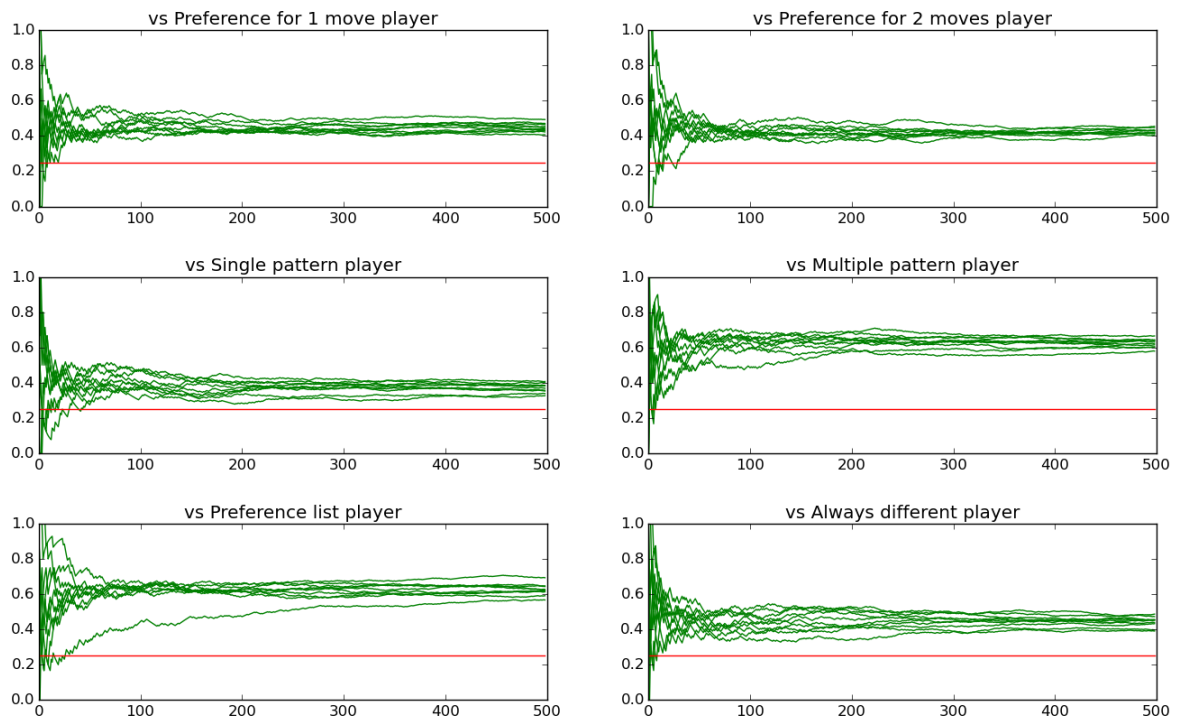


Figure 5.3: N-gram with time evaporation performance

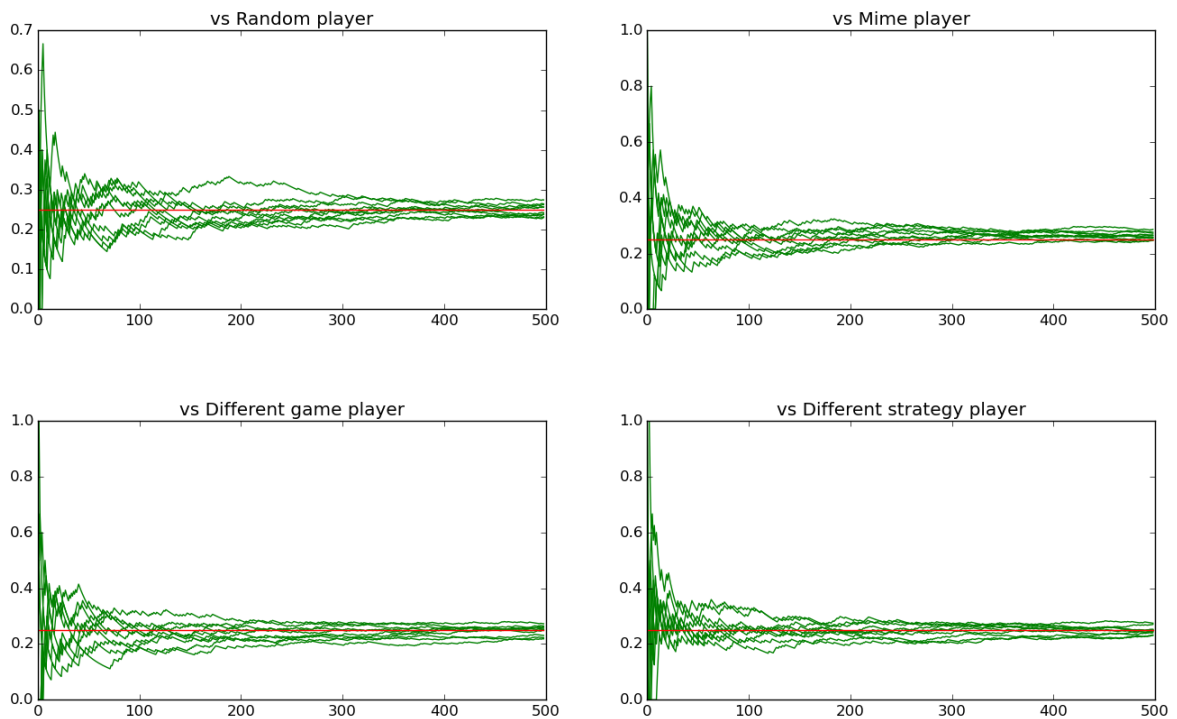


Figure 5.4: N-gram with time evaporation performance (continuation)

Chapter 6

Reinforced AI

The previous model achieved good performance, but with this model, we've tried another method to learn the player's strategy and we'll check if it produces good performance.

6.1 Model

This AI uses the same tree (see Figure 3.1) as the N-gram, but a small modification has been brought to it. Each node now stores an additional variable, that we call *gain* and is used to know the gain associated with that node.

A game is simply a path in the tree. At the end of the game, if we win, we reward the path by multiplying all the values of the nodes on the path by 2. If we lose the game, we penalise the path by dividing the value of the nodes on that path by 2. Initially, the gain is set to 1.

Anytime we need to assign some probability to our move, we will go at the right place in the tree and assign the probability of playing an action like this : $P[a] = \frac{\sum Gain_a}{\sum Gain}$
Gain_a represents the sum of the gains for an action *a* that can be reached from the current node. *Gain* represents the sum of the gains of all the subnodes of the current node.

As a reminder, the tree contains both the player's action and the AI's action on its branches, but we only want probabilities on AI's actions. So we sum together the gains of the nodes reached by the branch having the same AI's action.

Obviously, we need some smoothing for actions never played (because an action never played doesn't exist in the tree). Here we choose the *add-one* smoothing.

6.1.1 Java constraints

A non-negligible point is the fact this model performs a lot of divisions and multiplications. That means we can have variable overflow.

The model is implemented in Java (because we use Android), and the variables don't overflow in our case. But what happens in Java is not better than an overflow. Indeed, in Java, when a double initialised to 1 is divided more than 1000 times by 2, it reaches 0. It means that if a node is really bad, its gain will reach 0, and it will be impossible to increase its gain. In the same

way, when a double initialised to 1 is multiplied more than 1000 time by 2, it reaches Infinity. It means that it will be impossible to decrease it.

It does not adversely impact the performance of this model (as we will see in the next section), but it is important to keep it in mind because it means a very bad node will remain bad, and a very good node will remain good.

With some experiments, we saw that playing more than 5000 games could cause this problem to arise. We don't actually have a solution but we don't consider it to be a major issue.

6.2 Results

Figure 6.1 and Figure 6.2 sum up the performance of the Reinforced AI against the players.

This model gives good results, as good as the previous ones. There are only some differences as for as variance is concerned. Indeed, if we compare variance in this model with variance in the previous model against the *Preference list player*, we notice this model has a smaller variance. Now if we compare the same model against the *Always different player*, it's the previous model that has the smaller variance.

In conclusion, this other way to learn is simple and efficient but as exposed in the Section 6.1.1, it is limited because we use Java language.

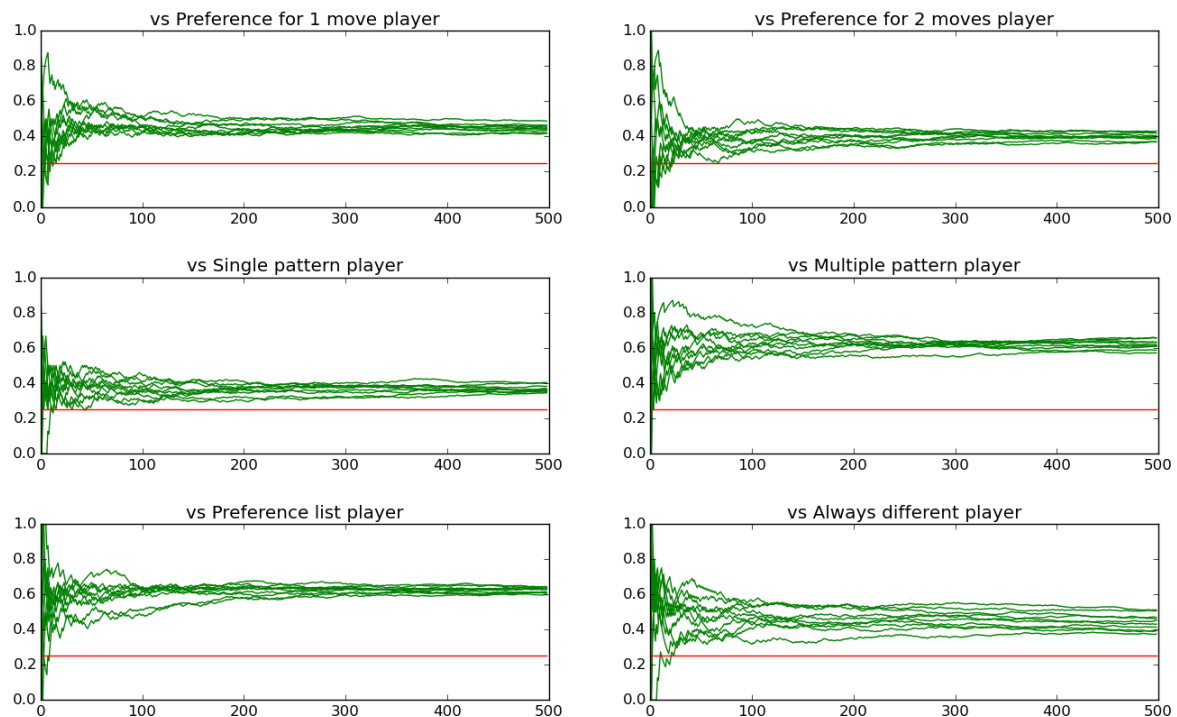


Figure 6.1: Reinforced AI performance

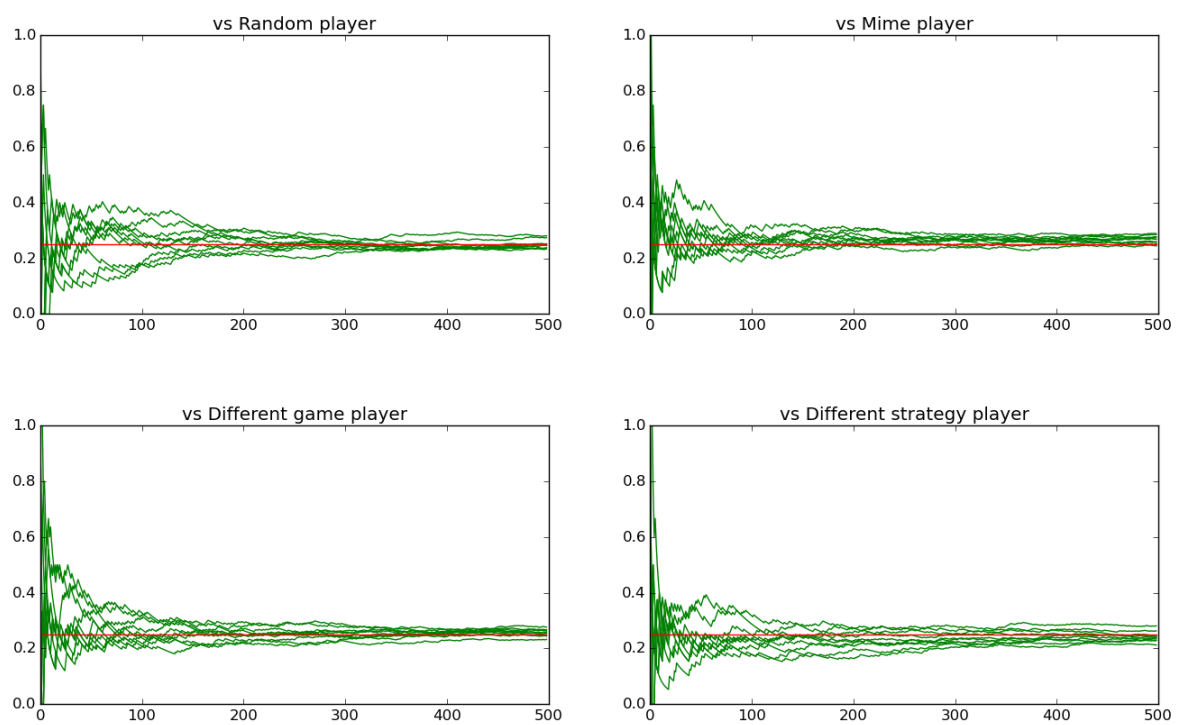


Figure 6.2: Reinforced AI performance (continuation)

Chapter 7

Conclusion

At this point, we will introduce a small conclusion about the first game. We will summarise relevant information and will conclude by explaining the reason why we stopped working on this game.

For all the AIs presented, we can notice one thing : they are all strong against the same players, and weak against the same players. Table 7.1 sums it up :

Learns against	Can't learn against
Always different player	Random player
Single pattern player	Different game player
Multiple pattern player	Different strategy player
Preference for one move player	Mime player
Preference for two moves player	
Preference list player	

Table 7.1: Players where learning is possible and impossible

To simplify the conclusion, we summarised a lot of information in Table 7.2. The leftmost column of this table contains the different models for this game. On top of the table, you'll find the different strategies against which we tested the models. In the cells, we thus have the win rate of the model (line) against the player (column) obtained over 10 runs (10 green lines).

First, we can observe that, when we play against a player that has a real strategy (like *Preference for 1 or 2 moves*, *Single pattern*, ...), the model can learn something from the games it played. Most of the models are particularly good at recognising specific patterns in the games with more than 50% wins against *Multiple pattern player*.

However, we can see that none of the AI can beat the *Different strategy player*, the *Mime player* and the *Different game player*, not even the N-gram with time evaporation AI. It can be explained by the fact the AIs take a lot of time to adapt. To counter the strategy of those three players, we need to take the last games played into greater consideration (greater than the N-gram with time evaporation AI).

Another thing we can notice is that none of the AI can beat the *Random player*. And it's an important thing because it means there is no better strategy than random strategy against the *Random player*. In other words, it means that :
There is no better strategy than playing randomly.

As we said in Section 1, we could continue along this way to learn the unconscious strategy of the player. However, when we examine the performance of our models against advanced players (good adaptation), we thought it would be hard to learn this unconscious strategy. Adaptation needs to be very efficient to learn this strategy.

Because of all the problems linked to this game, we chose to redirect this thesis on a game where more strategies could be learnt. This will allow us to do more advanced researche in Machine Learning.

	Players										
Models	Random	Prefer 1 move	Prefer 2 moves	Prefer- ence list	Single pattern	Multiple pattern	Different game	Always different	Mime	Different strategy	Mean
N-gram	0.235	0.391	0.320	0.510	0.317	0.498	0.247	0.205	0.243	0.227	0.319
Balanced N-gram	0.246	0.328	0.263	0.437	0.285	0.443	0.238	0.232	0.209	0.239	0.292
Evapora- tion N-gram	0.239	0.440	0.422	0.608	0.325	0.610	0.230	0.436	0.253	0.227	0.379
Reinforced AI	0.233	0.430	0.300	0.612	0.349	0.610	0.246	0.429	0.248	0.224	0.373
Best	Bal- anced N-gram	Evapora- tion N-gram	Evapora- tion N-gram	Rein- forced AI	Rein- forced AI	Evapo- ration N-gram and Re- inforced	N-gram	Evapo- ration N-gram	Evap- ora- tion N- gram	Bal- anced N-gram	Evapo- ration N-gram

Table 7.2: Summary of results for first game

Part II

A more strategic game

Chapter 8

Game overview

8.1 Rules

This new game is a betting game opposing two players. It features 5 actions on which the players can bet. When the game begins, each player has the same amount of money, i.e. 10 coins. At each round, both players bet on an action. They can bet on the same action but they are unaware of the other player's action. It's later explained why it's complex to know if the player wins an action or not. When the round is over, the actions played become unavailable for the rest of the game. A game is finished when there are no more available actions or if a player runs out of money. As there are 5 actions and as players can bet on the same action, there will always be 5 rounds in a game at the most.

The 5 actions are : 1, 2, 3, 4 and 5. Action 1 generates 1 coin, action 2 generates 2 coins, and so on. Betting on an action is called a move. We represent them as a tuple like this : (action, bet).

In a round, you can obtain three different results according to the moves of the two players, but in all cases, the players lose their bets :

1. Different actions (see Figure 8.1) :

In this case, as the left player (the purse on the left) didn't bet on the same action as the right player (the purse on the right), both players win the amount of their actions. The left player's move is (2,1). It means that he bets 1 coin on action 2. So he loses 1 coin and wins the amount of his action, i.e. 2 coins. In the end, his total gain is 1 coin. The right player's move is (4,1). It means that he bets 1 coin on action 4. So he loses 1 coin and wins the amount of his action, i.e. 4 coins. In the end, his total gain is 3 coins.

2. Same action but different bets (see Figure 8.2) :

Here, both players bet on the same action. So, the player with the highest bet wins the action. In this case, the left player played (3,2) while the right player played (3,1). As the left player's bet is the highest, he wins the action. In the end, the left player loses his bet (2 coins) but wins the amount of his action (3 coins), so he wins 1 coin. The right player loses his bet (1 coin) and wins nothing, so he loses 1 coin.

3. Same action and same bet (see Figure 8.3) :

Here, as both players have the same move (same action and same bet : $(3,2)$), nobody wins the amount of the action, but they both lose their bets. As they both bet 2 coins, they both lose 2 coins.

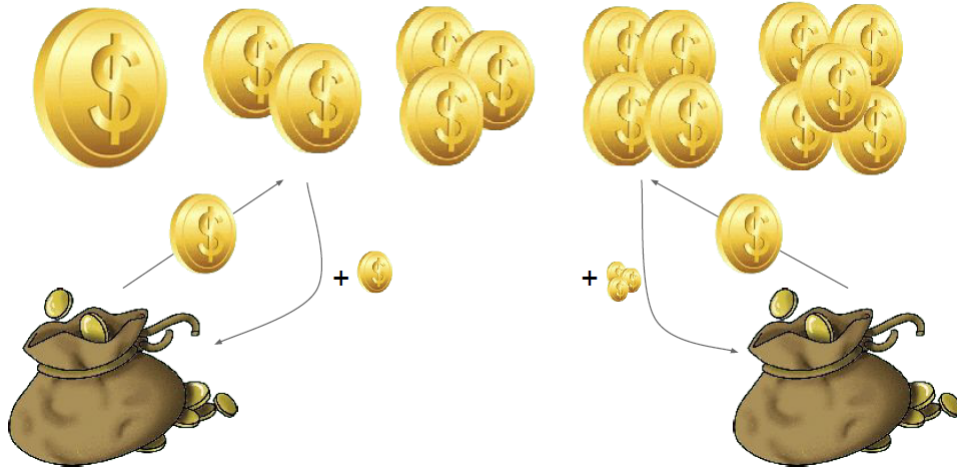


Figure 8.1: Round with different actions

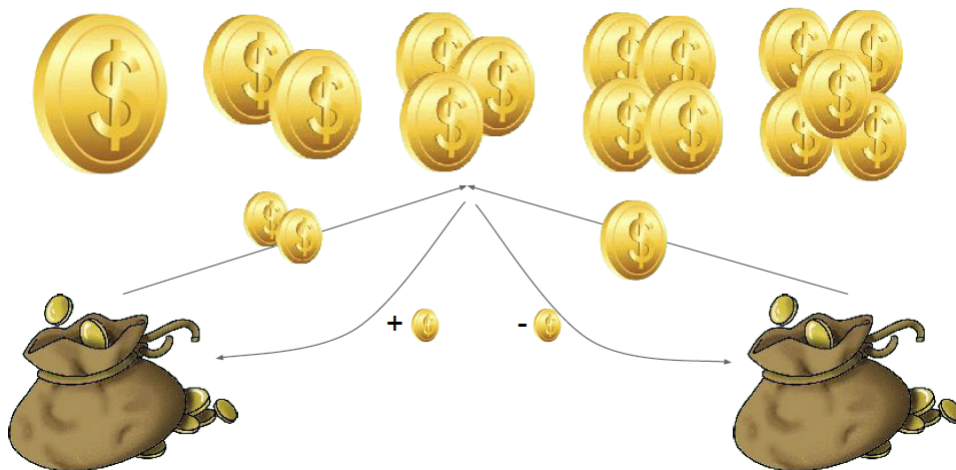


Figure 8.2: Round with same action but different bets

8.2 Data of the game

In this game, the data we'll have to deal with is the following :

- Money : each player start with 10.
- Actions : 1, 2, 3, 4, 5.
- Bets : from 1 to the player's maximum amount of money.
- Move : an action-bet pair (a,b) that a player performs.

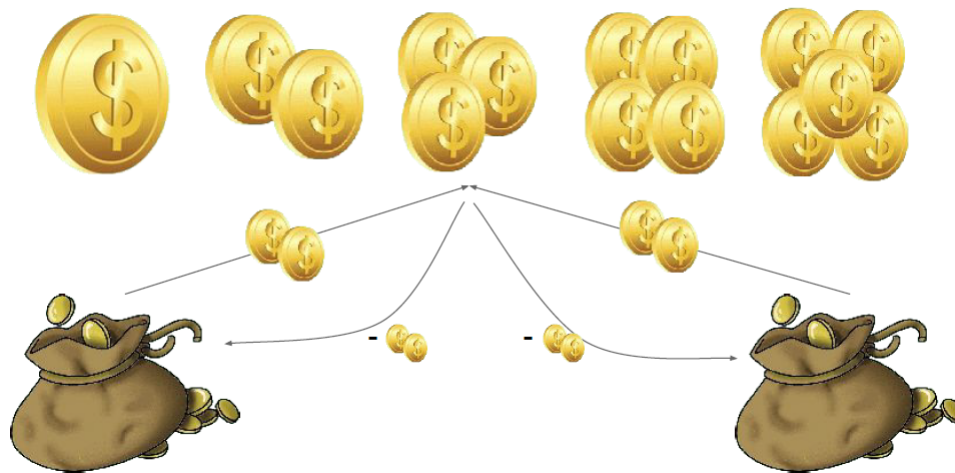


Figure 8.3: Round with same action and same bet

- Round : a pair of moves, one for each player.
- Game : a sequence of 1 to 5 rounds.

As in the previous game, we store the raw data in a database, because we want to be able to collect a lot of information in order to build up our models at best. Table 8.1 shows to table for storing the rounds. The table for the games is the same as for the previous game, illustrated in Table 1.4.

Rounds					
Game ID	Round nbr	Player1Choice	Player1Bid	Player2Choice	Player2Bid
1	0	5	7	3	8
1	1	2	7	4	3
1	2	1	2	1	3
2	0	5	3	3	3
2	1	4	3	2	1
2	2	1	1	1	1
3	0	4	9	1	2
3	1	2	5	5	1

Table 8.1: Rounds table in database

8.3 Example of game

Table 8.2 illustrates an example of this game. In this table, the three first columns show the state of the game and the two last ones show the moves played in that situation. The first column contains the available actions. Money P1 and Money P2 contain respectively the money of player 1 and player 2. Move P1 contains the moves played by player 1 and Move P2 the ones played by player 2. Recall that a move is a pair consisting of an action and a bet.

Actions	Money P1	Money P2	Move P1	Move P2
1 2 3 4 5	10	10	(5,1)	(5,3)
1 2 3 4	9	12	(3,1)	(4,3)
1 2	11	13	(2,2)	(2,2)
1	9	11	(1,2)	(1,1)
-	8	10	-	-

Table 8.2: Example of game

First, both players bet on action 5, player 1 bet 1 and player 2 bet 3. This is the point 2 explained in Section 8.1. Player 1 thus lose is bet and player 2 wins 2 coins ($5 - 3$). Then the available actions are 1, 2, 3 and 4 because 5 was played. In this situation, the players played on different actions. Player 1 chose action 3 and player 2 action 4. This is the first case of Section 8.1. Each player wins their action minus their bet : $3 - 1 = 2$ for player 1 and $4 - 3 = 1$ for player 2. After that, actions 3 and 4 are removed, only 1 and 2 remains. In this case, both players played the same move (same action and same bet). This is the third case presented in Section 8.1. In the event of a tie, they both lose their bet and win nothing. Finally we arrive in the situation where no action is left. As player 2 has more money (10) than player 1 (8), player 2 wins this game.

Chapter 9

Performance testing

The performance in this game is based on the win, lose and draw rates. Given the main objective for a model is to win the most games, the higher the win rate, the better the performance. A model is better than a player if its win rate is higher than the player's win rate i.e. the loss rate of the model. Ideally we would like that the model wins against all the players that we will introduce below.

In order to train our models, we make them play 1000 games against the different players quoted hereunder (200 games for each players). In that way, they have an opportunity to learn some good and bad moves. Moreover it simulates noise in the database; in fact, we don't expect the same player to play so many games.

9.1 Graphs

In all the plots presented in *Results* sections of the following chapters, we've drawn the win rates of the analysed AI in green, the win rates of the opponents (the player) in red and the draws in orange.

9.2 Pseudo players

This section will present all the players we've created in order to test the AIs. In this game, playing randomly is a very bad strategy because some moves are obviously bad. Since humans know when they play well or not, human moves will be motivated and thus we don't need to simulate an unconscious strategy. That's why, unlike the players of the first game, the players of this betting game do not play their strategy with a certain degree of probability unless specified in the description of the player.

9.2.1 Random player

It is the simplest player we can create. It is given a list of possible moves and it picks one at random.

9.2.2 Positive random player

This player plays randomly choosing among the available moves that won't make him lose money : bringing a positive or null gain. For example it won't play (3,5) because even if it wins the action, it will lose 2 coins (3 - 5). We are talking about the following moves : (1,1), (2,1), (2,2), (3,1), (3,2), (3,3), (4,1), (4,2), (4,3), (4,4), (5,1), (5,2), (5,3), (5,4) and (5,5). Obviously, if some moves are not possible, it doesn't consider them.

9.2.3 BALC player

BALC stands for Best Action and Lowest Cost. This player plays as following : for each round, it will take the best action (the one which brings the most money) and will bet as little money as possible, i.e. 1 coin. For example, for the first round, it will always play move (5,1). This player will be useful to test if the models can learn from a player with a fixed strategy.

9.2.4 BAPG player

BAPG stands for Best Action and Positive Gain. The idea of this player is the same as *BALC player* : it plays the best action. But here the bets are not fixed to 1 but goes from 1 to the value of the selected action. For example, at the first round, it can play one of the following moves at random (5,1), (5,2), (5,3), (5,4), (5,5). If the best action is 3, it will play (3,1) (3,2) or (3,3).

9.2.5 SAB player

SAB stands for Same Action and Bet. This player picks a remaining action at random and always bets the same value as the value of the action. Its possible moves are then (1,1), (2,2), (3,3), (4,4) and (5,5). Once again, it depends on its money.

9.2.6 Switch strategy player

This player uses the players presented above except for the *Random player*. It selects one player at random and every N games, it picks a new player at random. N is a user specified value, we usually use 200 when testing an AI on 2000 games.

Chapter 10

N-gram based on wins

In the previous game, we saw that the N-gram with time evaporation model gave us some good results. That's why we first decided to re-use it directly on this betting game. However, this game offers many more combinations in the number of moves : this causes the tree to be much bigger than in the previous game. To obtain a valid assessment of the probabilities, we need to have a significant number of games in each node. The consequence is that, in this game, we will need many more games than in the previous one. Since the time evaporation discards games after some time, we decided not to implement it first because we needed all the available games to have significant probabilities. That's why we first tried with the simple N-gram model, the same as in Chapter 3.

10.1 Model

Even if it is the same model, the tree structure is not exactly the same. Figure 10.1 represents this "new" tree we call Game Tree. We still have the different combinations of the player's moves on the branches, typically $(M1, M2)$ where $M1$ is the move of player 1 and $M2$ the one of player 2. The difference stands in the nodes. Instead of storing a simple number, we store the wins, the losses and the draws that AI achieved by playing the moves on the branch from the root to that node.

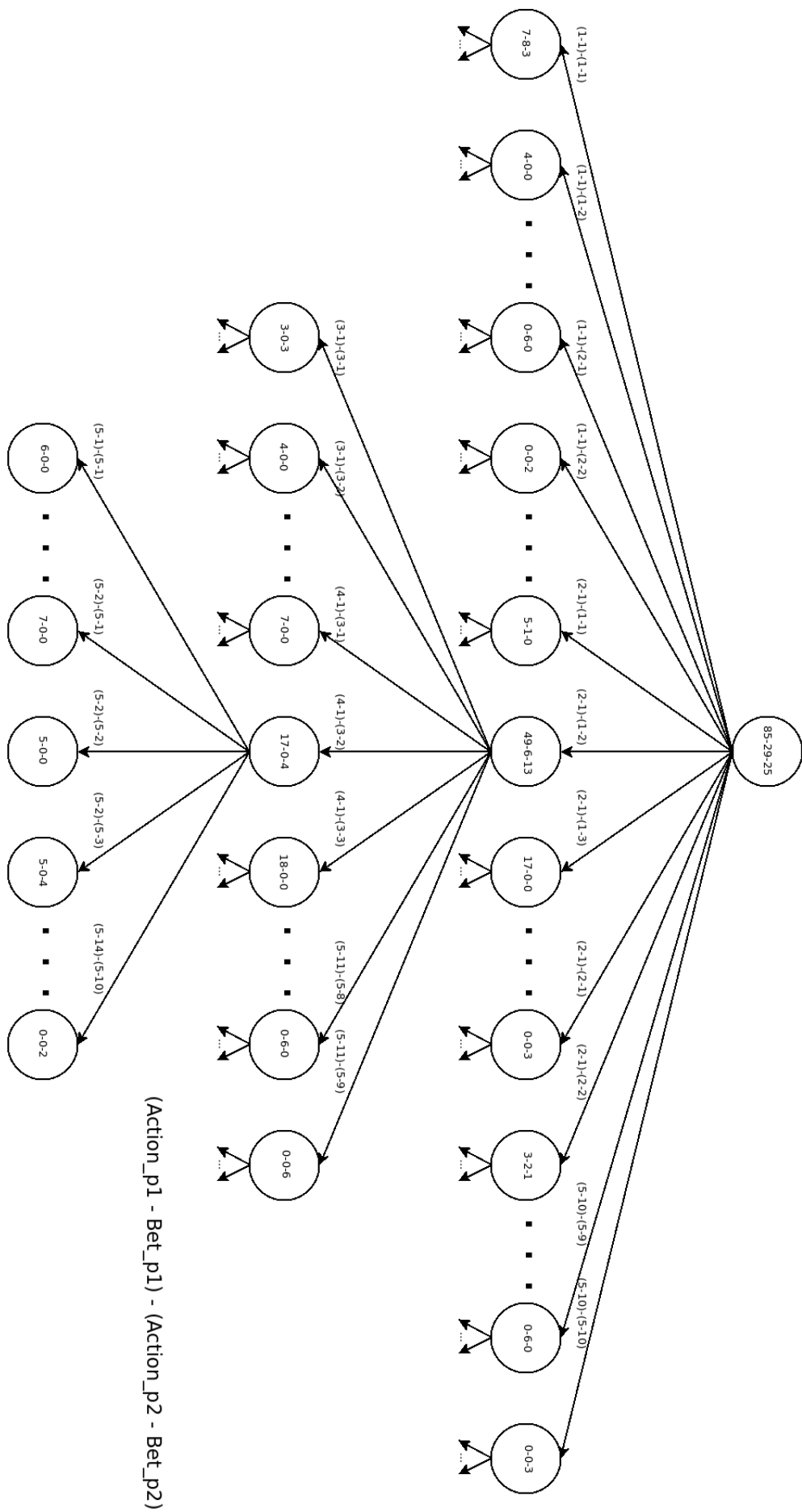


Figure 10.1: Game Tree

June 5, 2016

10.1.1 Choosing the move

In this section, we will describe how this model chooses the action to play. The first thing to do when the AI must select a move is to go to the correct location in the Game Tree corresponding to the state of the game (i.e. the moves played by both players in the current game). When choosing the move, two possible cases can happen :

1. In case the node corresponding to the state of the game doesn't exist :

It means that we are in a case that the model never encountered before and thus it doesn't have any information on how to play. That's why it assigns the same probability to each possible moves. It is equivalent to playing randomly.

2. In case the node corresponding to the state of the game exists :

In that case, we will count the number of wins for each moves we've already played. Each move we've already played corresponds to one or more children (remember that a branch is a pair containing the moves of player1 and player2). Then, we will divide this sum by the total number of wins (this total is stored in the current node). This is the probability of playing a move that we have already seen.

10.1.2 Some words about this model

As we said previously, there are a lot of possible combinations with this game. Indeed just for the first level of the Game Tree, both players have 50 possible moves : 5 actions multiplied by 10 coins. As we have two players, we have $50 \times 50 = 2500$ combinations for the first round. Because of the tree size, we have to play a lot of games to populate all this tree, and even more games to learn how to play the game, and even more games to learn the opponent's strategy.

In this game, all the possible moves are not good to play. For example, move (1,10) at the beginning is never good. Even if it guarantees to win action 1, it entails a huge loss of money for a very small reward. The problem of this model is that it takes a lot of time to learn that simple fact.

Another problem with this model is that the win rate that is used to assess the probabilities is not an efficient method. It's due to the fact that all the moves have almost the same probabilities for a pretty long time. The model has to play a lot of games to understand that a move leads to a win.

10.2 Results

We will now expose the performance of this model against different players. Actually, this model doesn't have good performance. As you can see on Figure 10.2, it has the same performance as the *Random player* and is beaten by all the other players. We can explain this thanks to the two following reasons :

1. The size of the tree : as explained in Section 10.1.2, the tree used for the data representation is so huge that the model takes too much time to populate it.

2. The win rate : this model learns the relevance of the various moves through the win rate. The problem is that the moves that are obviously bad are not quickly identified as such with this model. Moreover the probabilities with the win rate doesn't vary quickly and as we store more and more games, the impact of the latest game drops gradually.

Another point is that we don't observe any learning. Given the two reasons above, we explain this by the fact that there aren't enough games. But we don't know how many games we need to see a beginning in the learning process. We tried to run the AI for 10000 games against *Random player*, but we've never seen any sign of learning (difference between good and bad moves). We didn't try to go further than 10000 games, because it appeared exaggerated for a simple smartphone game.

Because we obtained very poor performance, we didn't try to improve this model. We directly tried to simplify the data representation and to change the way to compute the probabilities (replace the win rate by something else).

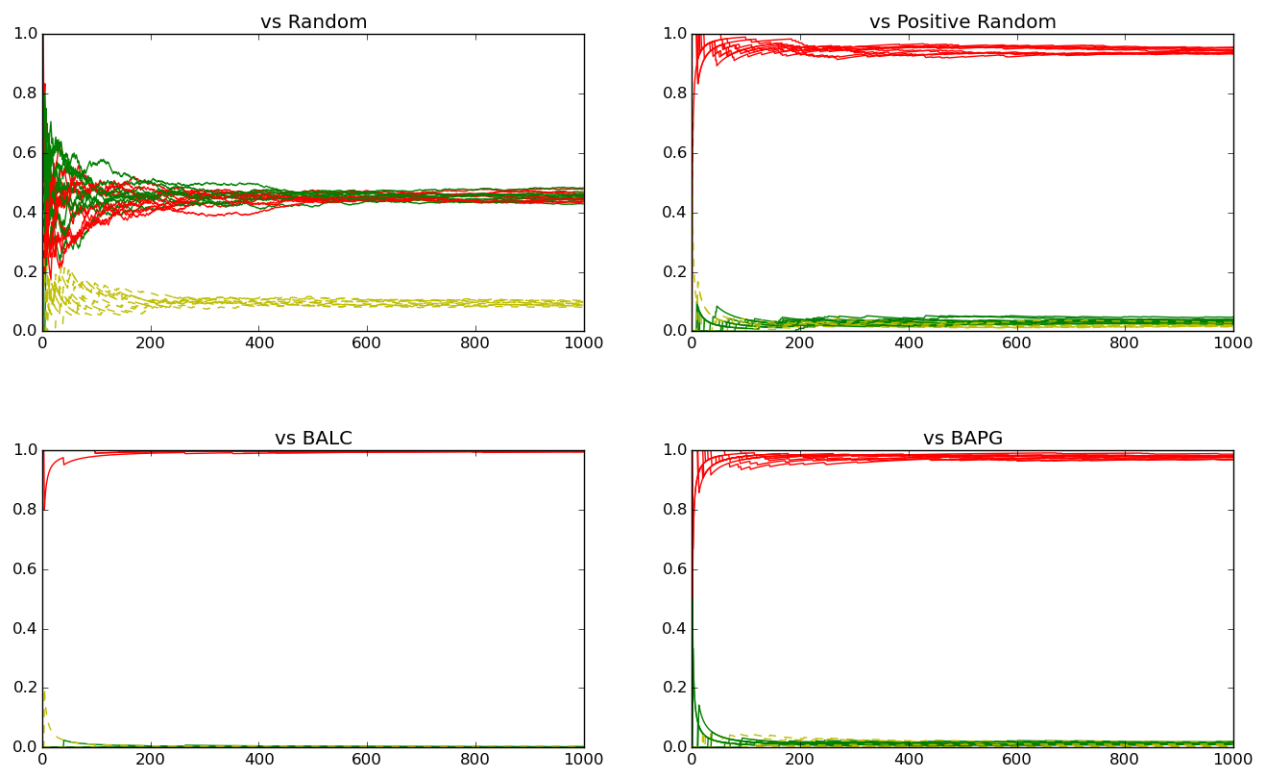


Figure 10.2: N-gram based on wins AI performance

Chapter 11

Matrix AI

As we saw in the previous model, focusing on the move of both player is not an efficient method to reach good performance. We decided to only focus on the AI's moves. To do so we created a model that would add weight on the moves, the heavier, the better.

11.1 Model

This model is based on a Matrix. Table 11.1 is an example of such a matrix. Remember that a move is a pair (action, bet). That's the information that will structure the matrix. The left side represents all the actions and the top cells represent the bets.

	1	2	3	4	5	6	7	8	...
1	100	51	1	1	1	1	1	1	...
2	212	43	1	1	1	1	1	1	...
3	267	88	32	4	1	1	1	1	...
4	461	214	74	42	5	2	1	1	...
5	827	543	276	211	37	9	2	1	...

Table 11.1: Example of matrix

The numbers in the cells allow us to know if a move is good or not. The higher the number, the better the move. Those numbers are obtained by updating the matrix at the end of each game. The update can be done in different ways :

Adding a fixed value

One of the simplest way actually consists in adding a fixed gain to all moves that were played if the AI won and in decreasing the value of the moves by the same fixed value if the AI lost. To choose the best value, we can perform a performance test on different values and keep the one that gives the best results.

Adding the result of the game

At the end of a game, for each move (a,b) played by the AI, we update the matrix[a][b] value by adding the score of the game : only taking the difference of money into account, we add (AI's money minus player's money). By doing so, the AI learns that the moves that were played in this game were bad if the AI lost and good if it won.

Adding a gain function

For each action played, we add the gain that a given action brought to the AI (in this game). We calculate the gain by (AI's action - AI's bet) - (player's action - player's bet) for each round in the game. By doing so, the moves that led the AI to a win are rewarded and the ones that resulted in a loss of money are penalised.

The worst way to update the matrix is to add a fixed value. This is simply because it adds no additional information other than the fact that the AI won or lost the game. We aren't provided with any clue as to how good or how bad it was.

That explains why adding the result of the game leads to a better model. The result (AI's money minus player's money) provides us with that information. If the AI wins with a great difference, that means that the moves the AI played were very good. Conversely, if the difference is very small, that means that the AI's moves were very bad.

But something else is missing. We don't know which of the AI's moves made the AI won, they are all rewarded in the same way. The difference is also influenced by how the opponent performed. If we manage to win with "bad" moves because the opponent played even worse, we will still record these moves as good ones because we won with them. This explains why the latest option achieves the best performance. In this way, we have a specific value for each action depending on its influence on the game. Furthermore, if a player often plays the same moves, the AI will increase its best response and learn the player's strategy. This account for the fact that we will only present the performance of this model with the *Adding gain function* option in the Result section of this model.

These update processes can lead to adding negative values and eventually cause some trouble in the computation of the probabilities. If this happens, we need to implement a normalisation step after the update. Here again, there is not only one way to do it and we will briefly present it :

Increase all

With this method, we add the absolute value of the minimum number in the matrix to all the cells of the matrix. For example, if the minimum value in the matrix is -6, we add 6 to every cell of the matrix so that there is no value under 0. The drawback of this technique is that the moves that are not played often (for example move (2,13) is only available if we won enough money in previous rounds) are often increased but not often decreased. So if that move becomes available

it has a high probability of being played.

Set to 1

In order to avoid the problem described above, we don't want to increase all the values. The other option is thus to set the negative value to a positive one. We thought about 0 but that would mean that once a move had a negative gain, we would never play it again. Instead, we want it to still have a low probability (because it might be a good action in certain cases). That's why we set the negative values to 1.

After testing these options, we can conclude that the *Increase all* is the worst method because it increases bad moves value. The set to 1 has another advantage : a bad move doesn't have to be double-checked because they have no way to gain value. It's that technique the model uses in the tests of the *Results* section below.

Then, the numbers in the matrix are used to compute probabilities. The probability of the AI to play action (a,b) is obtained by :

$$P[(a, b)] = \frac{matrix[a][b]}{\sum_{(i,j) \in posMoves} matrix[i][j]} \quad (11.1.1)$$

Where *posMoves* is the set of every possible move (i,j) in the current situation of the game.

For example, let's assume that only actions 1 and 2 remain available and the AI has 3 coins left. The AI will thus compute the probabilities for every possible actions : (1,1), (1,2), (1,3), (2,1), (2,2) and (2,3). The lower part of the equation is the same for all the actions and is equal to the sum of the values in the matrix for these actions : $matrix[1][1] + matrix[1][2] + \dots + matrix[2][3] = 100 + 51 + 1 + 212 + 43 + 1 = 408$. The probability of the AI to play (1,1) is thus : $P[(1,1)] = \frac{matrix[1][1]}{408} = \frac{100}{408} = 0.245$. The same goes for the other moves, we obtain $P[(1,2)] = 0.125$, $P[(1,3)] = 0.0025$, $P[(2,1)] = 0.520$, $P[(2,2)] = 0.105$ and $P[(2,3)] = 0.0025$.

11.2 Results

For the test, we chose the update and normalisation function that appeared to be the best, that is *Adding a gain function* and *Set to 1 the negative values*. The performance of this model is shown in Figure 11.1 and Figure 11.2. As you can see, this model performs way better than the previous one, in particular against the *Random player* where it wins almost every time. However, against the other players this is not good enough because it can't outperform the *BALC player* and *BAPG player*. A fixed strategy, as the one of the *BALC player*, should be easy to learn. When we see the performance against the *BAPG player*, we can't even expect that the AI will be able to be as good as that player.

Despite these bad results against those players, the performance against the *Switch strategy player* is not as bad as we could have expected (see Figure 11.2). This might indicate that this AI is better against a player that switches its strategy than against a player that always plays the same one.

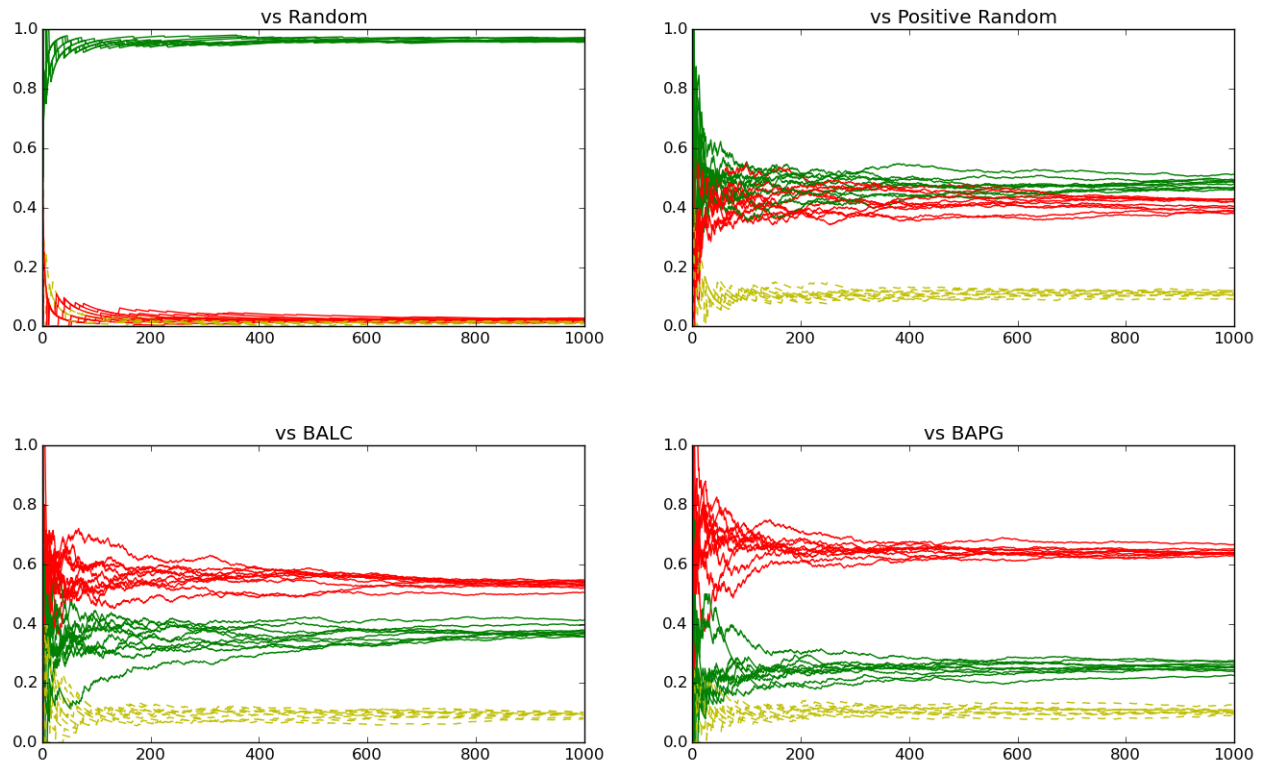


Figure 11.1: Matrix AI performance

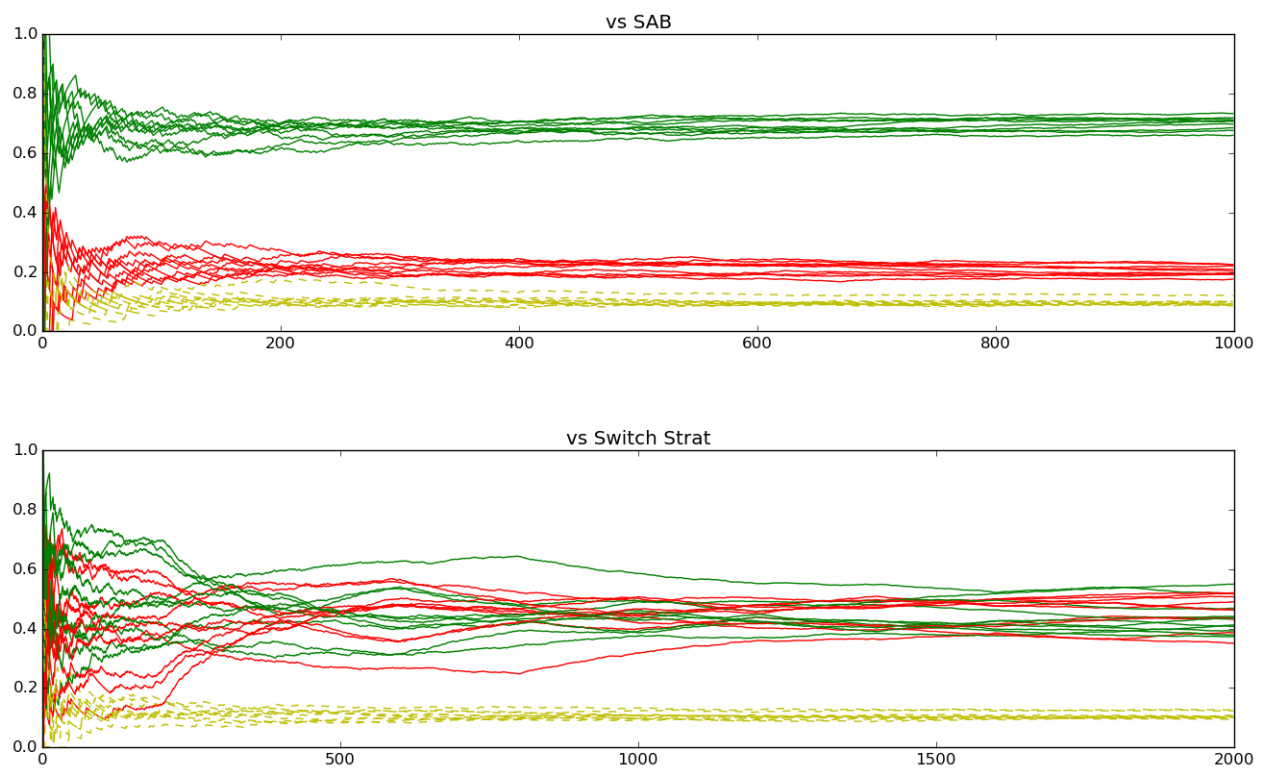


Figure 11.2: Matrix AI performance

Chapter 12

Opponent Tree AI

Limiting the data taken into account in the model provided us some encouraging results with the previous model. Another way to change the data representation that would be lighter than the first tree (represented in Figure 10.1) is to only take the opponent's moves into account. After a bit of thinking, we realised something : *once we know the opponent's next move, we can easily decide which move is the best to counter it.*

12.1 Model

This model will thus try to foresee to opponent's next move and play the move giving the best result to counter him. We must solve 2 problems for this model :

1. Finding a way to predict the opponent's next move.
2. Finding the move that maximises the result against the opponent's next. move

12.1.1 Finding the move beating another

If we are 100% sure that the opponent will play move $m_2 = (a_2, b_2)$, a move that maximises our gain of money (giving a positive or a null gain) can be easily found among our possible moves. The gain for player 1 is defined as below according to the moves of the two players :

$$Gain(m_1, m_2) = \begin{cases} (a_1 - b_1) - (a_2 - b_2) & \text{if } a_1 \neq a_2 \\ (a_1 - b_1) - (0 - b_2) & \text{if } a_1 = a_2 \text{ and } b_1 > b_2 \\ (0 - b_1) - (a_2 - b_2) & \text{if } a_1 = a_2 \text{ and } b_1 < b_2 \\ 0 & \text{if } a_1 = a_2 \text{ and } b_1 = b_2 \end{cases} \quad (12.1.1)$$

Where :

$m_i = (a_i, b_i)$ = move of player i

a_i = action of player i

b_i = bet of player i

For example : let's assume that action 5 is the only one available, and both players have 5 coins. If we are 100% sure that the opponent will play move (5,1), we can compute the gain

for all our possible moves against that move. In this case, our possible moves are : (5,1), (5,2), (5,3), (5,4) and (5,5).

$$\text{Gain}((5,1), (5,1)) = 0$$

$$\text{Gain}((5,2), (5,1)) = (5 - 2) - (0 - 1) = 4$$

$$\text{Gain}((5,3), (5,1)) = (5 - 3) - (0 - 1) = 3$$

$$\text{Gain}((5,4), (5,1)) = (5 - 4) - (0 - 1) = 2$$

$$\text{Gain}((5,5), (5,1)) = (5 - 5) - (0 - 1) = 1$$

Here, it is clear that the best move against (5,1) is (5,2) as the gain is the highest in that case.

12.1.2 The Player Tree

The problem is that we cannot be unquestionably certain that the opponent will play a given move. However, we can predict, as in the previous game, which move the opponent will play with a certain probability. To do so, we use a tree structure that we call the *Player Tree*.

This tree is represented in Figure 12.1. It stores all the opponent's moves from all the previous games. There are three numbers in each node : the first number represents the number of the opponent's wins, the second represents the number of the opponent's losses, and the last represents the number of draws. Each branch represents a move. For example, if we follow the central branch in Figure 12.1, we can see that if we begin a game with the opponent's move being (3,2), we know from the previous games that the opponent won 59 games, lost 27 and drew 15 times.

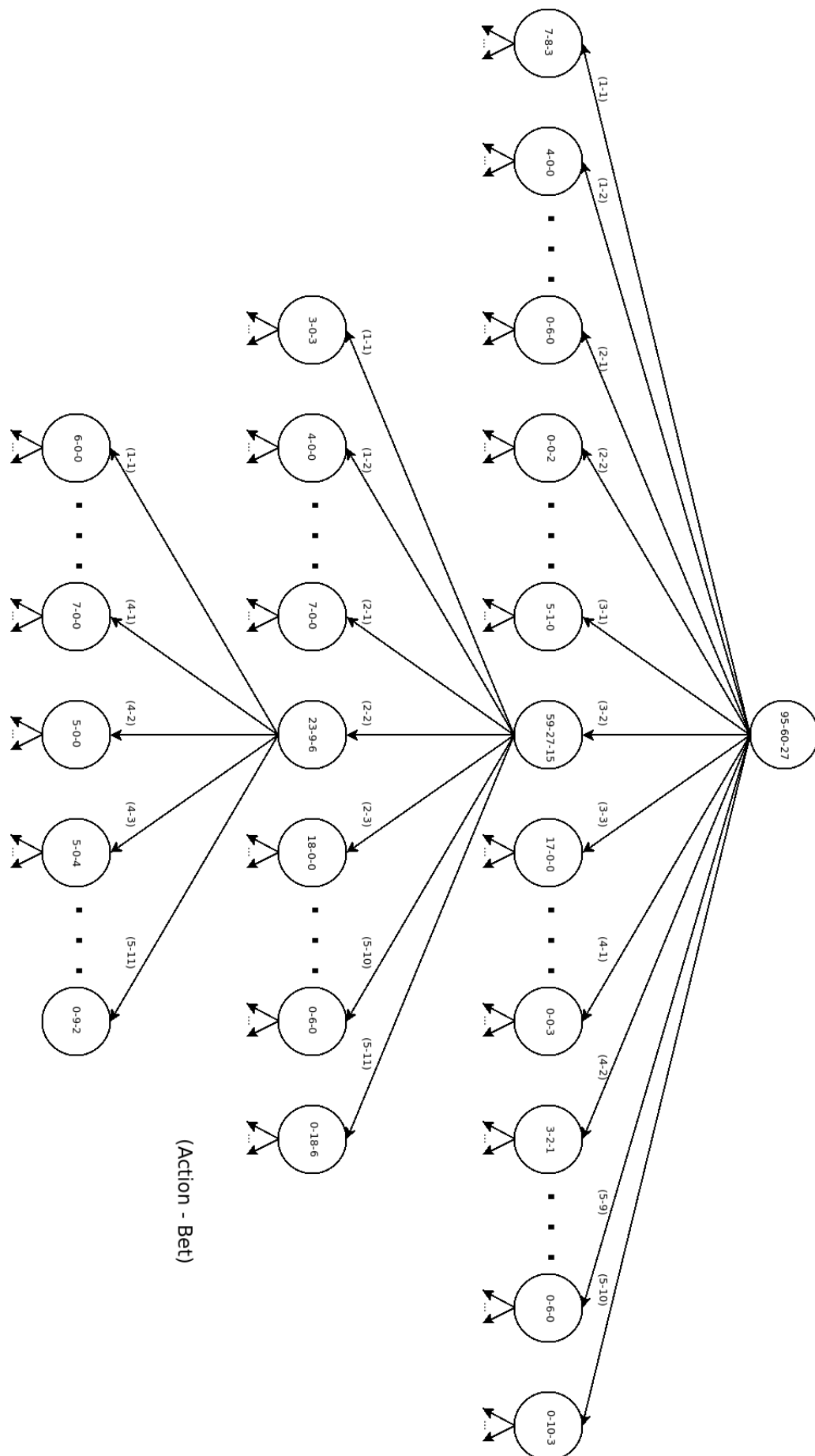


Figure 12.1: Player Tree

Thanks to the Player Tree, the counts of the opponent's moves for different sequences of actions are accessible to us, and we can assign probabilities to them. We can thus make a N-gram on the opponent's moves. For example : thanks to the first node in Figure 12.1, we can know there are $95 + 60 + 27 = 182$ previous games. And we can say move (3,2) has been played $59 + 27 + 15 = 101$ times. We can simply assume that move (3,2) will be played by the opponent with a probability equal to $\frac{101}{182} = 0.55$.

However, this tree is a subset of the data representation because we don't take AI's moves into account. Yet these moves may impact the opponent's moves. An example will be clearer. Consider the small Player Tree given in Figure 12.2.

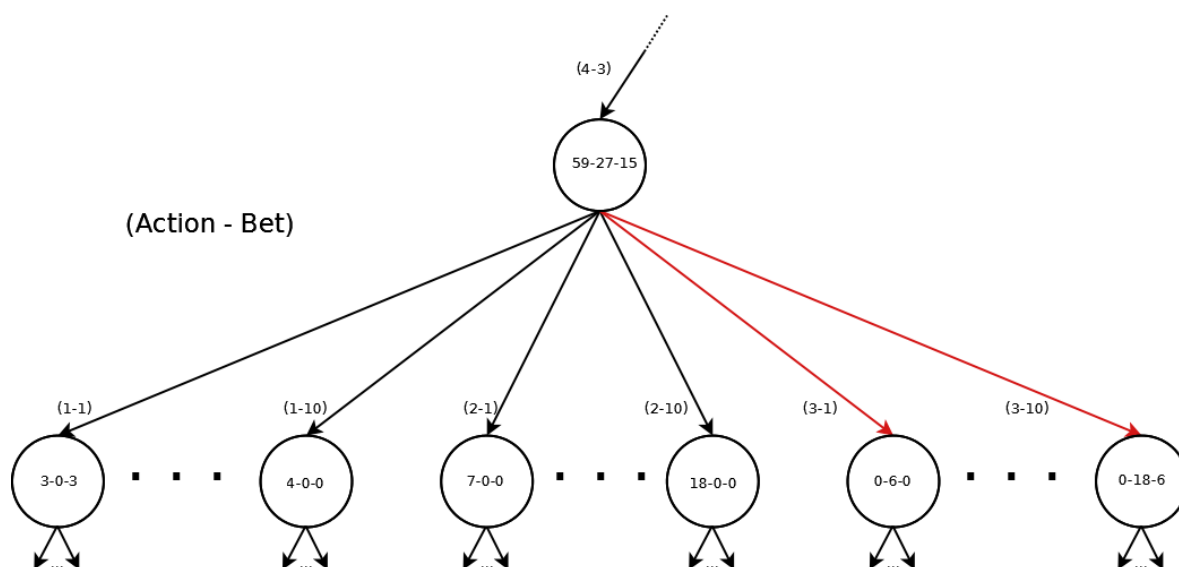


Figure 12.2: Player Tree problem

All the nodes are possible if we consider that the following sequence of actions has already been played :

Opponent's moves	AI's moves
(5,1)	(5,2)
(4,3)	(4,2)

Table 12.1: First history

That means that both the opponent and the AI respectively played (5,1) and (5,2) at the first round, subsequently they played (4,3) and (4,2). The problem is that if we change the AI's sequence of moves, the same nodes (of Figure 12.2) are not necessarily equally possible. Consider this new history :

As action 3 has been played, it's no longer possible to consider it in the Player Tree of Figure 12.2 (that's why those branches are in red).

It is also possible not to take some nodes into account because of a difference of money. In the first and the second history, there was no difference of money. But suppose this third history

Opponent's moves	AI's moves
(5,1)	(5,2)
(4,3)	(3,1)

Table 12.2: Second history

occurred :

Opponent's moves	AI's moves
(5,1)	(5,2)
(4,3)	(4,4)

Table 12.3: Third history

After this history, the opponent's amount of money has changed. Indeed, after the first and the second history, the opponent's money totalled $10 - 1 + 1 = 10$, but after the third history, his money totalled $10 - 1 - 3 = 6$. Consequently, we must only take the nodes reached with a bet equal or lower to 6 into consideration.

If we are not aware of this, we could lose consistency in the probabilities. But, even if we talk a lot about this problem, the solution is pretty simple. Instead of dividing the count of the possible nodes (in the previous example with Figure 12.1, it was 101) by the count of the parent node (182 in our example), the divisor must be the sum of the counts of the possible nodes : $18+4+\dots+6+2+\dots+101+\dots$ (note that the result won't necessarily be 182). This way, the problem is solved.

12.1.3 Computing the AI probabilities

Here, we will see a simple example to really understand how this AI works. Let's consider that after some rounds both players have only 2 coins and that only action 1 and 2 are available. So the Player Tree for the opponent at that level is represented in Figure 12.3. All the nodes represent the number of times the corresponding moves (the branches) have been played. In this Figure, the numbers in the nodes are the sum of the numbers of wins, losses and draws that are actually stored in the nodes. From this, the AI computes the probability of each of the opponent's move. It's represented in Figure 12.4.

For each move, we will determine which of the AI's moves are the best, that is the one maximising its gain (see Equation 12.1.1). This is represented in Table 12.4. In this table, the AI's moves are represented on the top and the opponent's ones are on the left. The number in the cells corresponds to the gain resulting from playing the AI's move against the opponent's move.

Note : for the opponent's move (2,2), we see that 2 moves yield the same gain. In this case, we'll chose the first move that we see, i.e. (1,1).

Then the AI will assign his move the probability of an opponent's move against his own move

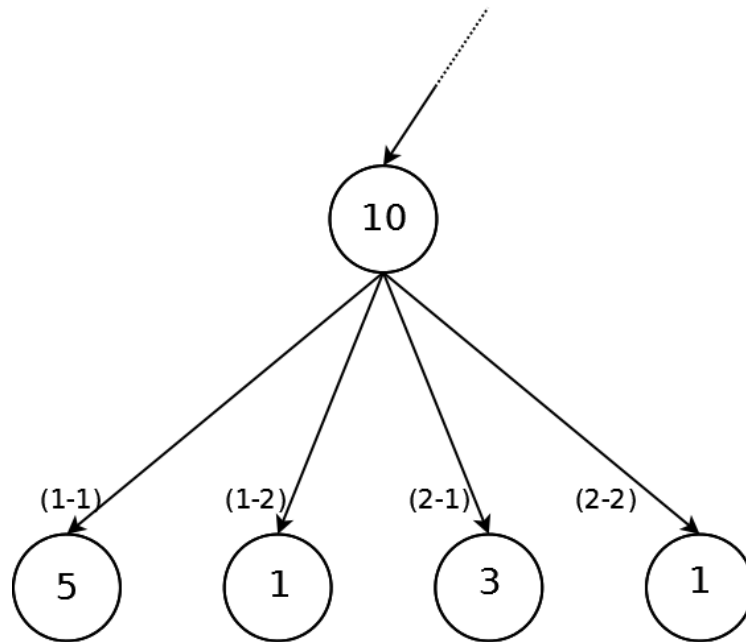


Figure 12.3: An example of Player Tree

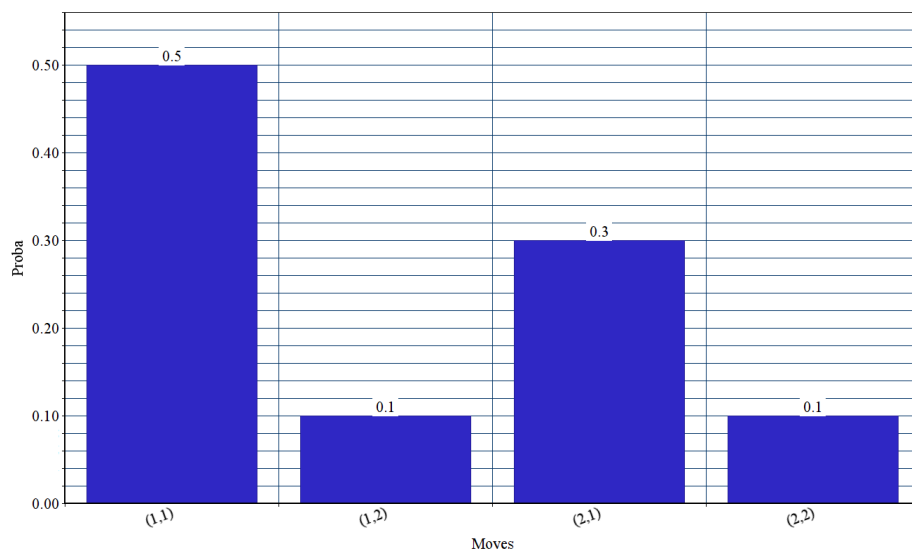


Figure 12.4: Probability for opponent's move

	(1,1)	(1,2)	(2,1)	(2,2)
(1,1)	0	0	1	0
(1,2)	0	0	2	1
(2,1)	-1	-2	0	1
(2,2)	0	-1	-1	0

Table 12.4: Gain computation - AI moves (top) and opponent moves (left)

will win. The probability formula :

$$P_{AI}[m] = \sum_{o \in \text{Countered}(m)} P_{opp}[o] \quad (12.1.2)$$

Where $\text{Countered}(m)$ is the set of opponent's moves countered at best by m (according to the gain function). Formally, it is defined as below :

$$\text{Countered}(m) = \{y \in \text{opponent moves} \mid (\arg \max_{x \in \text{AI moves}} \text{Gain}(x, y)) = m\} \quad (12.1.3)$$

From Table 12.4, it appears that move (2,1) beats both moves (1,1) and (1,2). It will thus take the probability of move (1,1) and the one of (1,2). So $P_{AI}[(2,1)] = P_{opp}[(1,1)] + P_{opp}[(1,2)] = 0.5 + 0.1 = 0.6$. (see Figure 12.4 for probabilities).

Move (2,2) beats (2,1). So $P_{AI}[(2,2)] = P_{opp}[(2,1)] = 0.3$. Move (1,1) also beats (2,2). So $P_{AI}[(1,1)] = P_{opp}[(2,2)] = 0.1$. As move (1,2) counters no move, it will not be played, and will have a probability of 0. Finally the probabilities for the AI moves are :

- $P_{AI}[(1,1)] = 0.1$
- $P_{AI}[(1,2)] = 0$
- $P_{AI}[(2,1)] = 0.6$
- $P_{AI}[(2,2)] = 0.3$

12.2 Results

The advantage of this model is that it directly realizes what are the better moves thanks to the gain function. Without any previous game, it can see that betting 10 on action 1 is obviously not a good move. This allows to directly win against the *Random player*. On the graphs (Figure 12.5 and Figure 12.7) we observe this model has globally good performance. Furthermore, a learning process is noticeable against some players.

Against the *Positive random player*, we see that this model adapts itself quickly (in the course of the first 100 games) and after this threshold its performance stabilizes, reaching its maximal win rate against this player.

When it plays against *BALC player*, which has a fixed strategy, it starts winning pretty early, and the more games we have, the more the AI wins. Clearly, some learning process against this player is noticeable.

Against the *BAPG player*, the time to learn the strategy is a bit longer than against the other players. But we observe some learning. As you can see in Figure 12.6, this model still learns after 3000 games but the speed at which it learns seems to decrease.

This model has no problem to beat the *SAB player*. This is explained by the fact that the *SAB player* doesn't have an "optimal" strategy : it can't win money, it can only make its opponent lose some money if it plays the same action.

Against the switch strategy player, we can see at some time that the green line decreases drastically. This is explained by the fact that this model is not as successful in learning the *BAPG player*'s strategy. So when the strategy switch to the *BAPG player*, the performance is impacted

but the win rate doesn't decrease linearly because the model still learns the *BAPG player's* strategy. The win rate thus decreases slower and slower. But globally the final performance is still good (between 0.58 and 0.78).

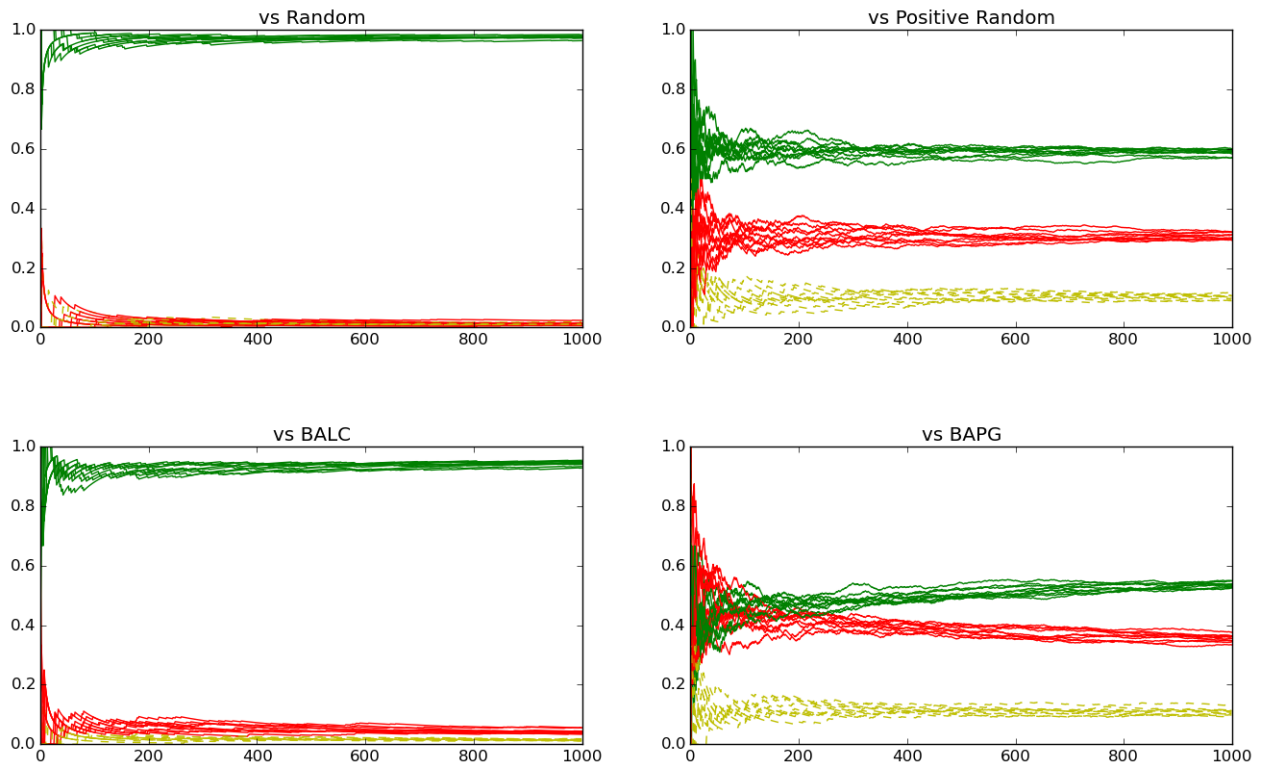


Figure 12.5: Opponent Tree AI performance

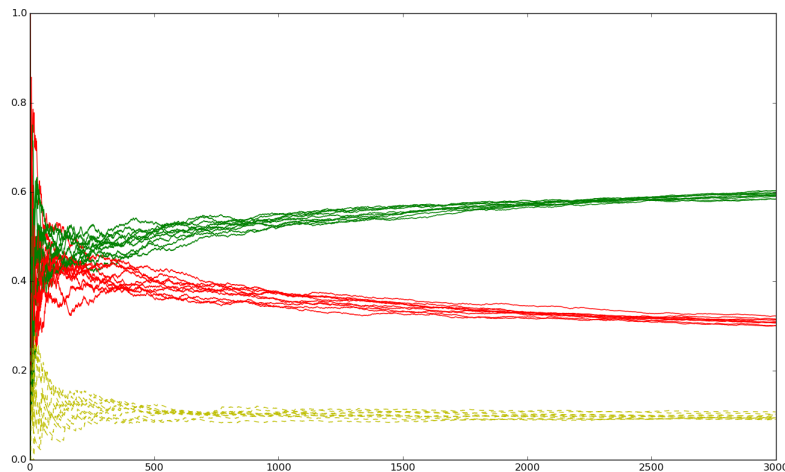


Figure 12.6: Opponent Tree AI performance vs BAPG (3000 games)

12.3 With a list of best moves

In Section 12.1.3, we said that in case more than one best move (same best gain) was available, the AI opted for the first one it encountered. In this extension, we use a list of the best moves.

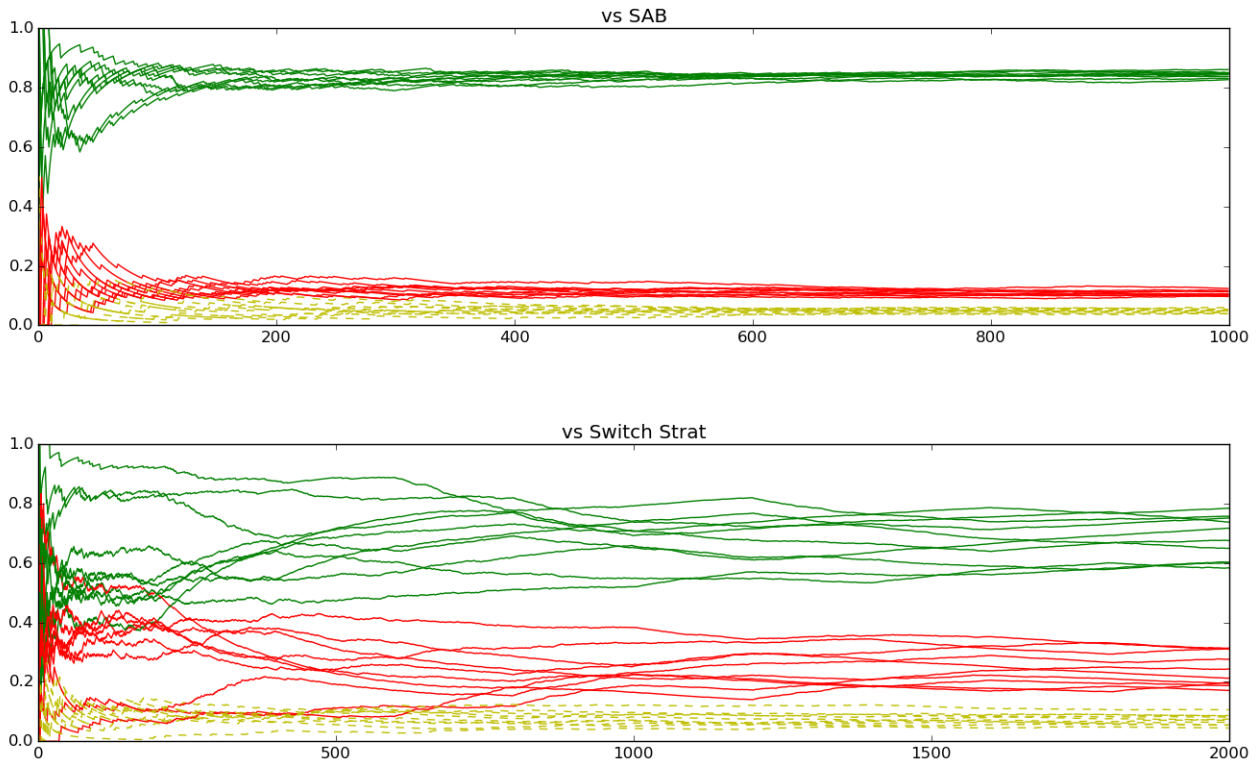


Figure 12.7: Opponent Tree AI performance

If we go back to the example in Section 12.1.3, we see that against the opponent's move (2,2), the AI moves (1,1) and (2,2) yielded the same gain.

In this variant, we put those moves in a list, and we play the list with the same probability as the one the opponent will have to play his move (2,2). So $P_{AI}[(1,1), (2,2)] = P_{opp}[(2,2)] = 0.1$. When we have to play the list, as we can play only one move, we will play randomly one element of the list. It amounts to assign each move of the list the probability of the list divided by its size. So $P_{AI}[(1,1)]$ and $P_{AI}[(2,2)]$ will receive $0.1/2 = 0.05$ each. The AI probabilities become :

- $P_{AI}[(1,1)] = 0.05$
- $P_{AI}[(1,2)] = 0$
- $P_{AI}[(2,1)] = 0.60$
- $P_{AI}[(2,2)] = 0.30 + 0.05 = 0.35$

In the example of Section 12.1.3, move (1,1) was played with a probability of 0.10. Because of this model, it is now played with a probability of 0.05. The same goes for move (2,2) : it was already played with a probability of 0.30. Thanks to this model, its probability is increased by 0.05.

12.4 Results with list

When we use the list in this model, we obtain pretty much the same performance as the model without the list. However, we still observe a small improvement against *BALC* player. Against

this player, we see that the model learns less easily but the win rate upper bound is slightly higher than in the previous model. Against *BAPG player*, we notice the same observation, it learns less quickly but ends up with a higher win rate in the end.

Now, if you look at the performance against the *Switch strategy player* in Figure 12.9, we observe this model has better performance than the previous one. Its win rate spans between 0.64 and 0.8 against the *Switch strategy player*.

We can conclude that this variant is slightly better than the basic version, but it does not improve significantly.

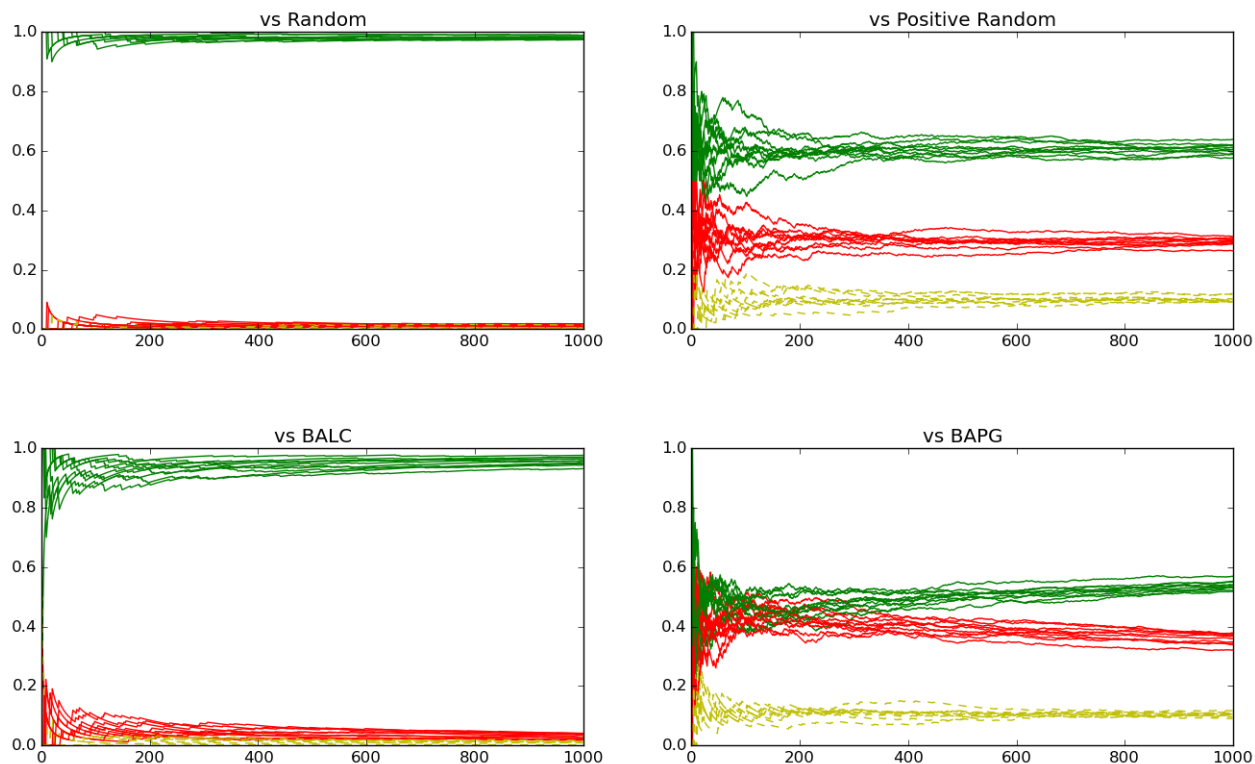


Figure 12.8: Opponent Tree AI with List performance

12.5 Smoothing the opponent tree AI

Up to now, for the Opponent Tree AI, we haven't used any smoothing technique. That means that all the moves that have never been played are assumed to have no chance to be played. However, this might not be true. It's not because a player never played move (4,4) that it could not play this move in this game. It could be interesting to assign a small probability to the unseen moves. The smoothing technique will be applied to the probabilities of the opponent's moves and not to the AI's moves.

There are multiple smoothing techniques that can be applied. We started with the simplest, the add-one method. This consists in adding 1 to the count of every move. That is, all the opponent moves have at least $\frac{1}{N_{games} + N_{moves}}$ chance to be played by the opponent, where N_{games} is the sum of wins, draws and losses of a node in the tree, and N_{moves} is the number of children

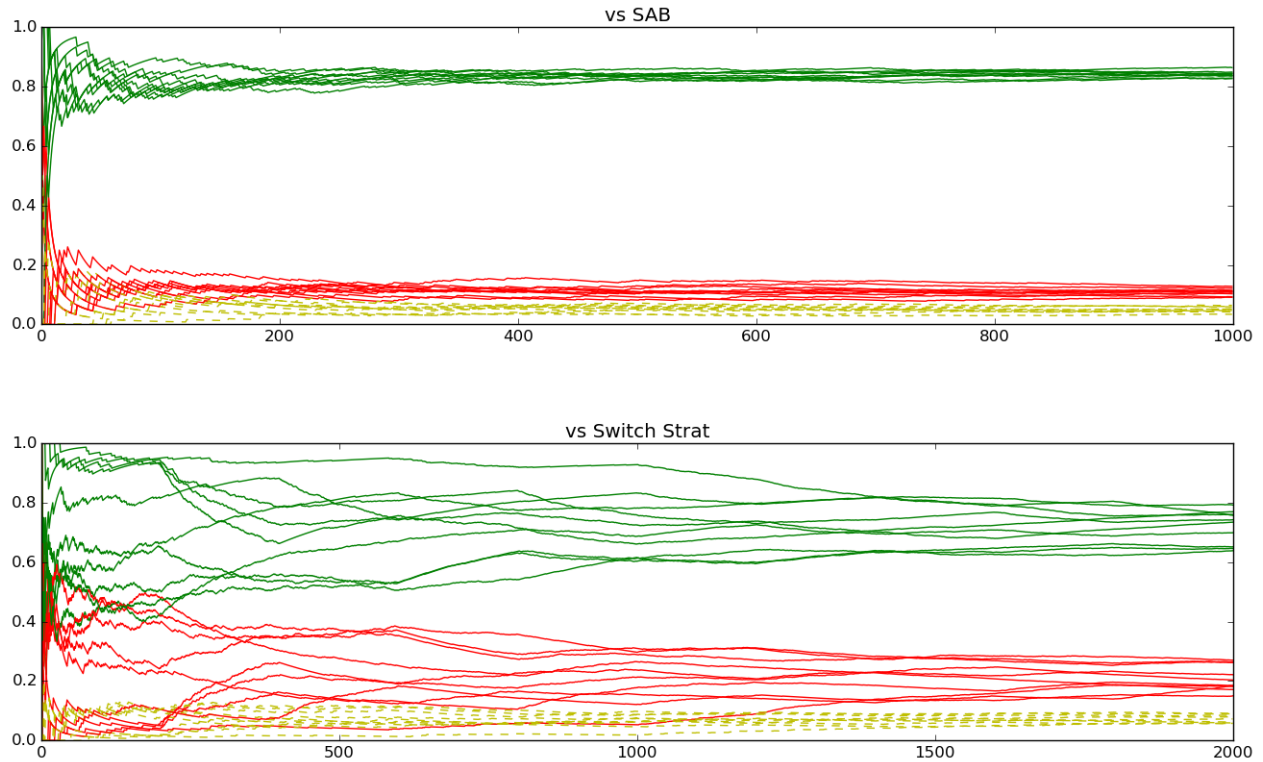


Figure 12.9: Opponent Tree AI with List performance

of that node. This smoothing is simple to implement but gives poor performance. We had poorer performance with this smoothing than without it.

We then implemented the **Backoff smoothing**. The idea behind this technique is, for every move, to add a portion of the probability obtained for the same move with smaller histories. To compute this smoothing, we use the following formula :

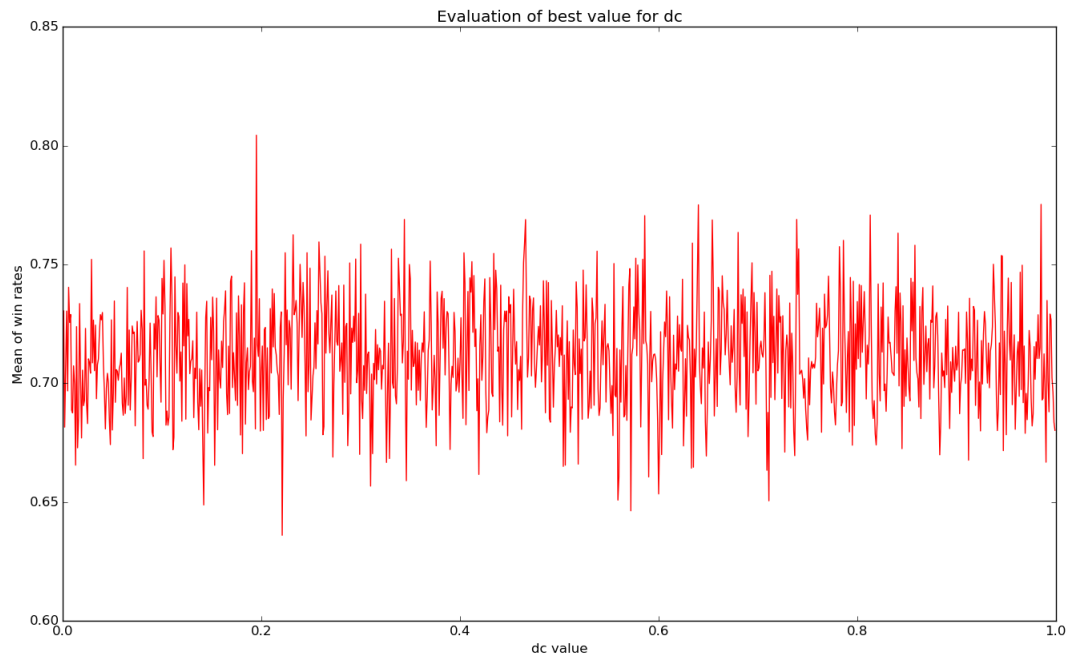
$$\hat{P}(w|h) = \begin{cases} \frac{C(h,w)-d_c}{C(h)} + \gamma(h)\hat{P}_{back}(w|h) & \text{if } C(h,w) > 0 \\ \gamma(h)\hat{P}_{back}(w|h) & \text{if } C(h,w) = 0 \text{ and } C(h) > 0 \\ \hat{P}_{back}(w|h) & \text{if } C(h) = 0 \end{cases} \quad (12.5.4)$$

This formula looks a bit hard but we will explain each part and establish a link with our game.

First, what we are computing with this formula, $\hat{P}(w|h)$, is an estimation (the hat) of the probability of observing move w after a sequence of moves h , called history.

Then $C(h,w)$ is the count of h 's followed by w 's : it is the number of times that the sequence of moves h was followed by move w . And $C(h)$ is the count of h 's : number of times the opponent played the sequence of moves h alone. d_c is a discount parameter. This parameter has a value between 0 and 1 and we set its value based on a performance evaluation over multiple values (we test each value and kept the one maximising the performance). In this case, the value of d_c maximising the win rate of the model is 0.195 as you can see in Figure 12.10. $\gamma(h)$ is a function taking the sequence of moves as parameter. It computes a probability mass discounted from seen events depending on d_c (see [8] for more information).

Finally, $\hat{P}_{back}(w|h)$ is the Backoff probability, that is the $\hat{P}(w|h)$ with a smaller history. This smaller history is the history without the first move of the sequence.

Figure 12.10: Evaluation of d_c

12.6 Results with smoothing

This model is an extension of the Opponent Tree AI with list (see Section 12.4). By comparing the results of Figure 12.11 with Figure 12.8 we can see that smoothing has the advantage of having a smaller variance. The comparison of Figure 12.12 and Figure 12.9 leads to the same observation. That means that adding smoothing to the model doesn't really add a gain in performance but makes the model more stable. By more stable, we mean the variance is smaller.

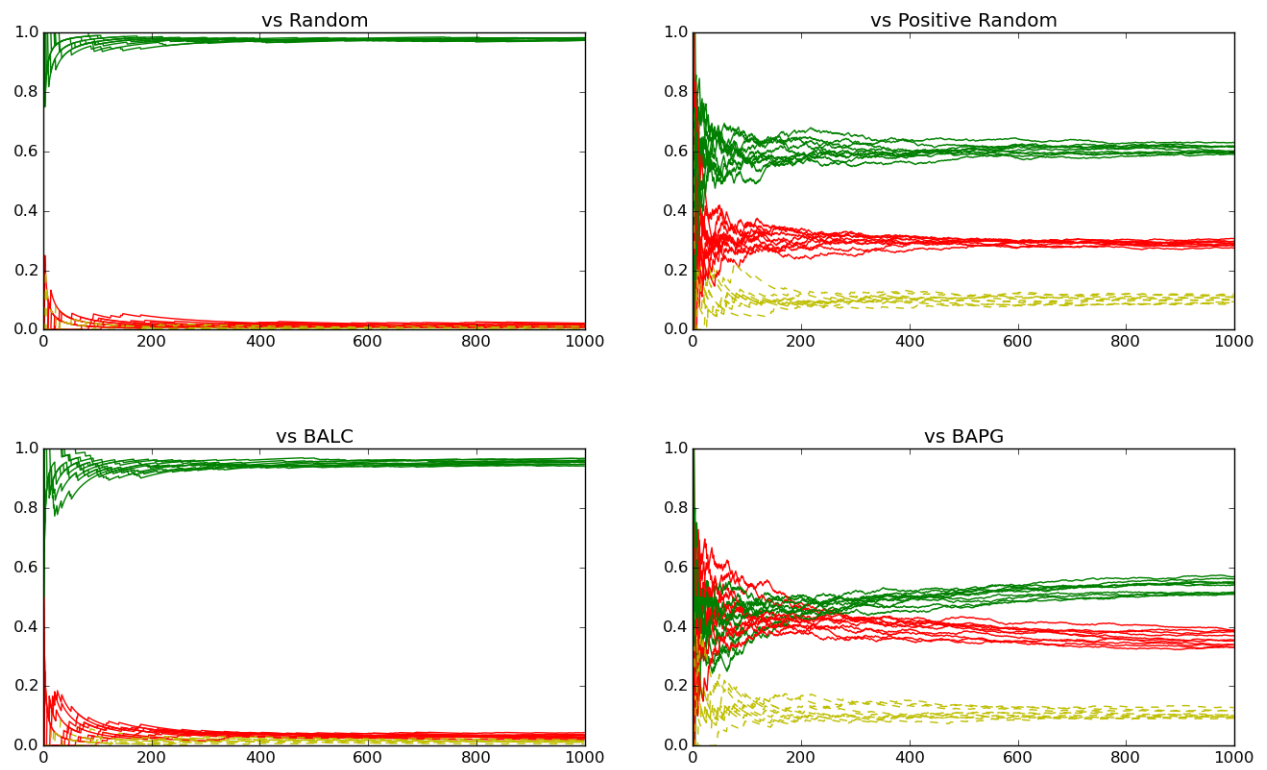


Figure 12.11: Opponent Tree AI with Smoothing performance

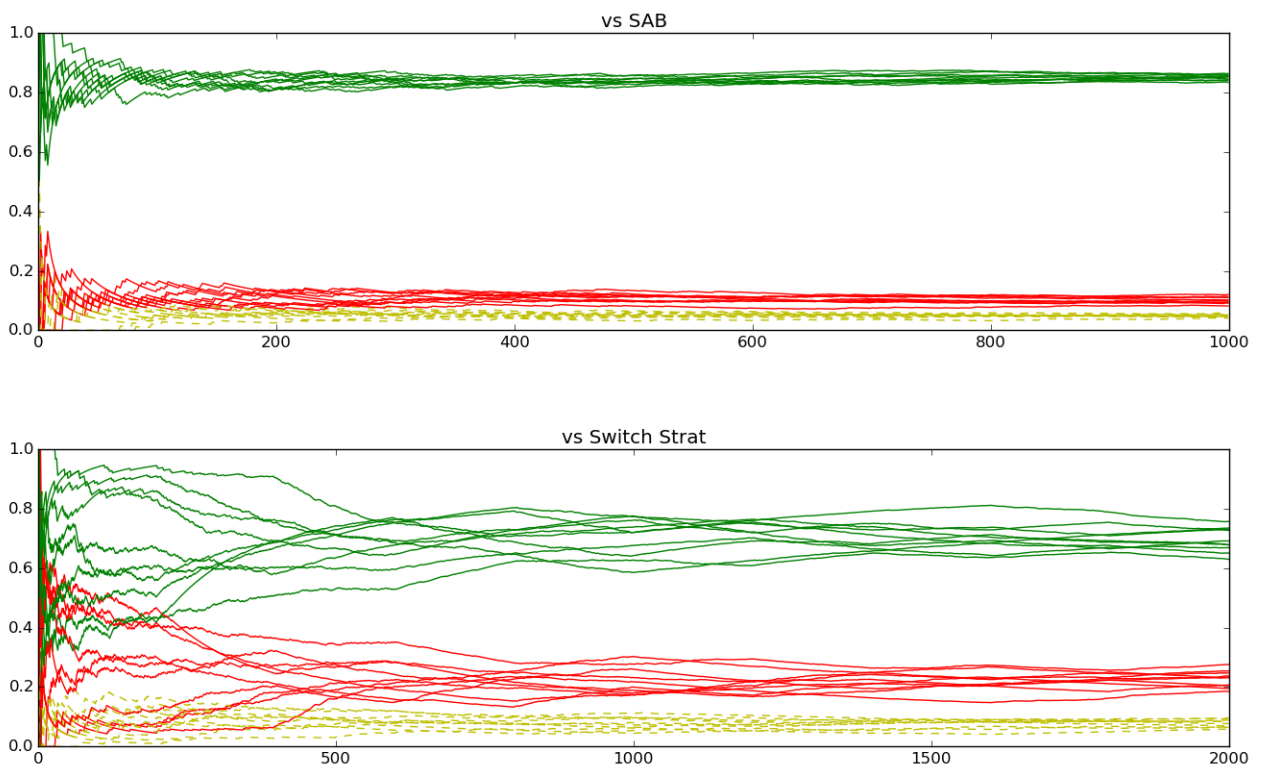


Figure 12.12: Opponent Tree AI with Smoothing performance

Chapter 13

Opponent Tree AI with win

After the previous model (see Section 12), we asked ourselves if it was possible to learn how to play the game without specifying the gain function, i.e. if it was possible to learn that playing move (1,10) at the beginning of the game was a bad thing. That's why we've created this model. With this model, we'll try to learn how to play to the game thoroughly.

13.1 Model

This model uses a Player Tree (see Section 12.1.2) like the Opponent Tree AI (see Section 12) to predict the opponent's moves. The novelty consists in the replacement of the Gain function by a structure allowing us to learn which move is better against another : the Dual Tree.

13.1.1 Dual Tree

Figure 13.1 illustrates this tree. This tree contains only two levels. The first level corresponds to all the opponent's possible moves (in red) while the second corresponds to all the AI's possible moves (in green).

As in the other trees, each node stores the number of wins, loses and draws according to the AI. So, even the opponent's nodes (from the first level) contain the AI's wins, losses and draws. But it's not a problem because we only care about the data from the AI's point of view.

This tree stores an history of moves and the results for each of them based on the opponent's moves. In other words, this tree should help the AI know which move is good (thanks to its number of wins, draws and loses) against an opponent's move.

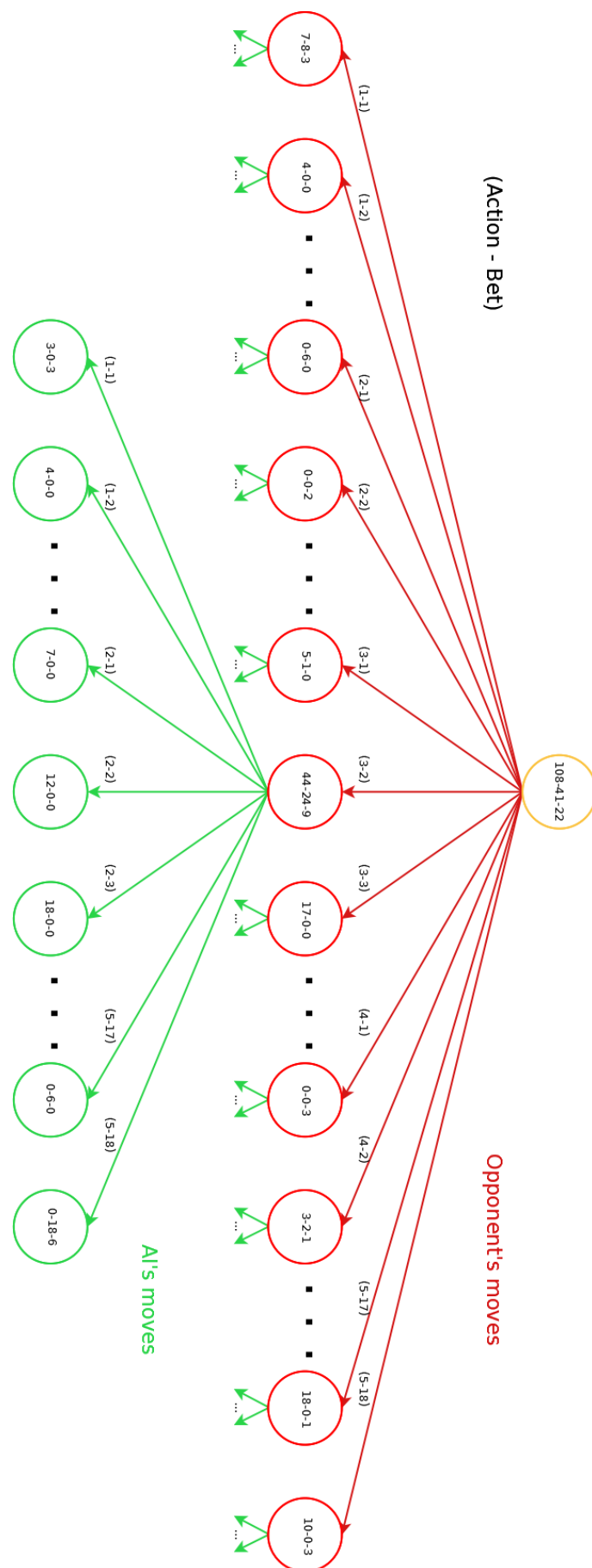


Figure 13.1: Dual Tree

Consider the simple example where we know the opponent will play move (3,2). In the tree (Figure 13.1), we go to the node in the first level corresponding to this move (it's the middle node). All children of this node are the AI's possible moves it can play. In order to know which move is the best, we look at a specific ratio that is explained in the next Section (13.1.2). Each node is assigned a ratio based on its wins, losses or draws to determine which ones are good.

13.1.2 Evaluating good moves for the AI

This model works pretty much as the Opponent Tree AI (see Section 12). It uses a Player Tree to assess the probability of each move the opponent may play. For each of these moves, we will try to find the best move to counter it, and assign the best move it the countered opponent's move probability.

Now, we will explain how to determine which move is the best against a specific move without knowing the Gain function (see Equation 12.1.1). To do so, we took the rationale of the N-gram based on wins (see Section 10), but we slightly improved it.

The first situation is when the node at the first level of the Dual Tree doesn't exist. In other words, it means we haven't seen this opponent's move yet. As the model has no information, it can't make any assumption and thus it plays randomly.

Now, if the node exists, we know that there is at least one child (because if the node exists, we've played against it). If we only consider the win rate (the number of wins of this node divided by the sum of the number of wins of each move), we could face a problem : in fact, in case all the possible moves we can consider in the tree only had losses and draws, but no wins, all the moves would have a ratio set to 0.

To solve this problem, we could consider using the draw rate. $Draw\ rate = \frac{number\ of\ draws}{number\ of\ games}$

The idea is to play the move that yields the most draws. But we can have the same problem as before. Let's imagine our moves had only losses, we would still have a ratio of 0 for each move. At this point, we need to consider the losses. The idea is to play the move with the fewest losses. So we define the loss rate : $Losse\ rate = \frac{1}{number\ of\ losses}$

The model computes the rate of each move as followed :

1. The model computes the sum of wins of our possible moves. If it's different from 0, it uses the win rate. Otherwise it tries point 2.
2. The model computes the sum of draws of our possible moves. If it's different from 0, it uses the draw rate. Otherwise it goes to point 3.
3. It uses the loss rate.

Once the ratio of each move is defined, it is finally possible to compute its probability.

13.1.3 The computation of the probabilities

Here, we will present how the AI computes the probabilities. As it is quite complex, we first thought it would be better to begin with an example represented in Table 13.1 and Table 13.2.

Let's assume only action 1 is available and both players have 3 coin. Table 13.1 represents this situation.

Probability	Opponent moves	AI moves	W, L, D	Rate	Probability
$\frac{3}{4}$	(1,1)	(1,1)	3, 10, 3	$\frac{3}{4}$	$\frac{3}{4} * \frac{3}{4} = \frac{9}{16}$
		(1,2)	1, 5, 10	$\frac{1}{4}$	$\frac{3}{4} * \frac{1}{4} = \frac{3}{16}$
		(1,3)	0, 3, 7	0	$\frac{3}{4} * 0 = 0$
$\frac{1}{5}$	(1,2)	(1,1)	1, 2, 3	$\frac{1}{8}$	$\frac{1}{5} * \frac{1}{8} = \frac{1}{40}$
		(1,2)	1, 4, 1	$\frac{1}{8}$	$\frac{1}{5} * \frac{1}{8} = \frac{1}{40}$
		(1,3)	6, 3, 1	$\frac{6}{8} = \frac{3}{4}$	$\frac{1}{5} * \frac{3}{4} = \frac{3}{20}$
$\frac{1}{20}$	(1,3)	(1,1)	0, 6, 3	0	$\frac{1}{20} * 0 = 0$
		(1,2)	1, 4, 3	$\frac{1}{2}$	$\frac{1}{20} * \frac{1}{2} = \frac{1}{40}$
		(1,3)	1, 3, 5	$\frac{1}{2}$	$\frac{1}{20} * \frac{1}{2} = \frac{1}{40}$

Table 13.1: Example of computing probabilities with the Dual Tree

The first column *Probability* expresses the probability with which the opponent will play the move represented in the second column. Against one opponent's move, the AI can play its three available moves : it expressed in the third column. The fourth column, *W, L, D*, stands for the Wins, the Losses and the Draws yielded by the AI's move when it is played against the opponent's move. Thanks to the fourth column, it is possible to compute the rate displayed in the sixth column. The last column is the computed probability of the AI's move according to its rate and the opponent's move probability.

If we take the first column, we know the opponent will play move (1,1) with a probability of 75%. Against this move, AI can play its three available moves :

- (1,1) with a rate of $\frac{3}{4}$.
- (1,2) with a rate of $\frac{1}{4}$.
- (1,3) with a rate of 0.

Instead of selecting the best move according to its rate, we decided to consider all the moves. To do so, we multiply the rate of each move by the probability of the opponent to play its move. That's why, in case the opponent plays move (1,1), the AI's move (1,1) has the probability of $\frac{9}{16}$ to be played.

Once we've done this for each opponent's possible move, we have to group the AI's moves probability together. Table 13.2 represents it.

In order to group the AI's moves probability, we just add the probability together. If we take the AI's move (1,1) as an example :

- It has a probability of $\frac{9}{16}$ to be played if the opponent plays move (1,1).
- It has a probability of $\frac{1}{40}$ to be played if the opponent plays move (1,2).

AI moves	Final probability
(1,1)	$\frac{9}{16} + \frac{1}{40} + 0 = \frac{47}{80}$
(1,2)	$\frac{3}{16} + \frac{1}{40} + \frac{1}{40} = \frac{19}{80}$
(1,3)	$0 + \frac{3}{20} + \frac{1}{40} = \frac{14}{80}$
Sum	$\frac{47}{80} + \frac{19}{80} + \frac{14}{80} = \frac{80}{80} = 1$

Table 13.2: Final probabilities

- It has a probability of 0 to be played if the opponent plays move (1,3).

We sum them together and obtain the final probability of $\frac{47}{80}$ to be played.

We can notice one thing : given that the opponent's moves probability and the rate are consistent (consistency property), this succession of operations keeps the consistency property. The last row of Table 13.2 proves it.

13.1.4 Some problems with this tree

As for the Player Tree, the data represented are a subset of all the data we have. Here, we don't care about which round we are in, and so we don't take the impact of the previous moves into account. For example, when the model searches which moves it can play, it will find some moves discovered in other rounds but not available for the current one.

So, we have a lot of moves in the Dual Tree but we must first check if we can actually play them. The solution to this problem is really simple. We just need to check if the move on the branch is contained in the list of the possible moves the AI received by the application. And as we do not consider all the nodes, we cannot disregard the consistency property.

13.2 Results

As this model doesn't have Gain function (see Equation 12.1.1), the performance is badly impacted. Indeed, we use a special rate (win, draw or loss rate), and as that special rate sometimes corresponds to the win rate, it's normal to have bad performance as previously seen in the first model. If we look at the first graph (against the *Random player*) of Figure 13.2, we see this model is better than the first model, but it doesn't beat this player thoroughly like the model of the previous Section (see 12.2). And if we begin with that kind of performance against the *Random player*, we can't expect it to perform any better against the other players. Indeed, if we look at the other graphs, we notice that this AI really has poor performance. That's why we didn't try to go further with the idea of learning the Gain function.

As a conclusion for this model we can notice that using Gain function is a really good advantage for a model. It allows the model to pick up of the rules of the game and consequently have an idea of what it has to do in order to win.

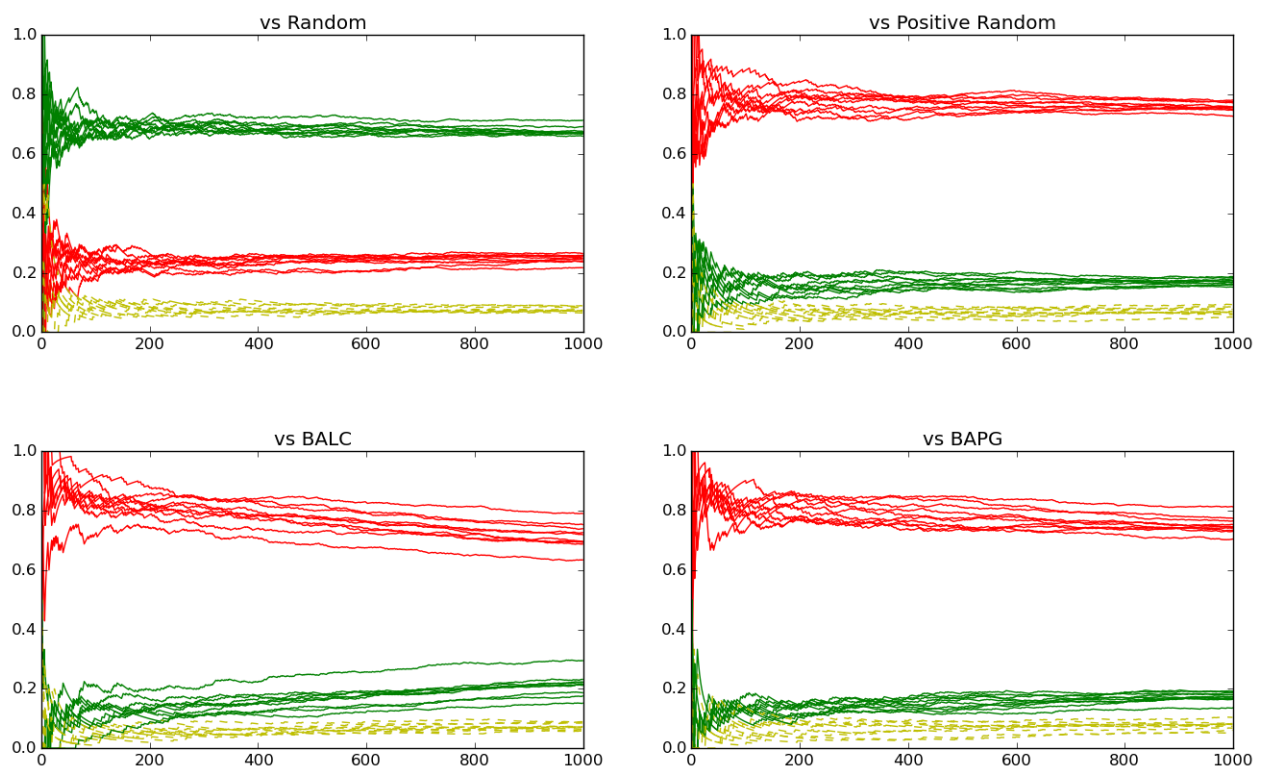


Figure 13.2: Win Opp Tree AI performance

Chapter 14

Opponent Tree AI with time evaporation

As we observed in the first game, using time evaporation on the N-gram model gave some good results. The N-gram with the opponent tree (Section 12) seems to be an efficient way to predict the opponent's moves. To go further with this model we will use time evaporation on the opponent's tree and see if it improves the performance of the model.

14.1 Model

This AI works as the Opponent Tree AI (see Section 12) to predict which moves the opponent will play and how to beat it. The difference lies in the size of the data considered and the way they are taken into account. Indeed, this model uses a parameter in order to know how far in the past it must go to predict the opponent's moves. In other words, it works as if it had a limited database containing only TTL game (see Chapter 5).

As TTL is a parameter, we have made a parameter evaluation on it to determine which value it should be assigned to. We can see the result in Figure 14.1. We notice that we have a high win rate when the TTL is equal to 94. As the neighbourhood is good on average, we decided to fix the parameter TTL to 94.

We have implemented this model in exactly the same way as for the Time evaporation N-grams of the previous game (see Section 5). As a reminder, we store a list of numbers inside the nodes of the Player Tree. Once a game is finished, we add TTL value inside the list of the nodes corresponding to the moves played during the game (in our case, TTL equals 94). Then all the values inside the list of each node will be decreased by 1. When a value reaches 0, we remove it from the list. And we only consider nodes with a number in the list. That means, the more recent the game is, the greater impact it has.

14.2 Results

This model has good performance. Indeed, if we look at Figure 14.2, we notice that it perfectly learns the strategy of *BALC player* after a few games and learns more quickly against the *BAPG*

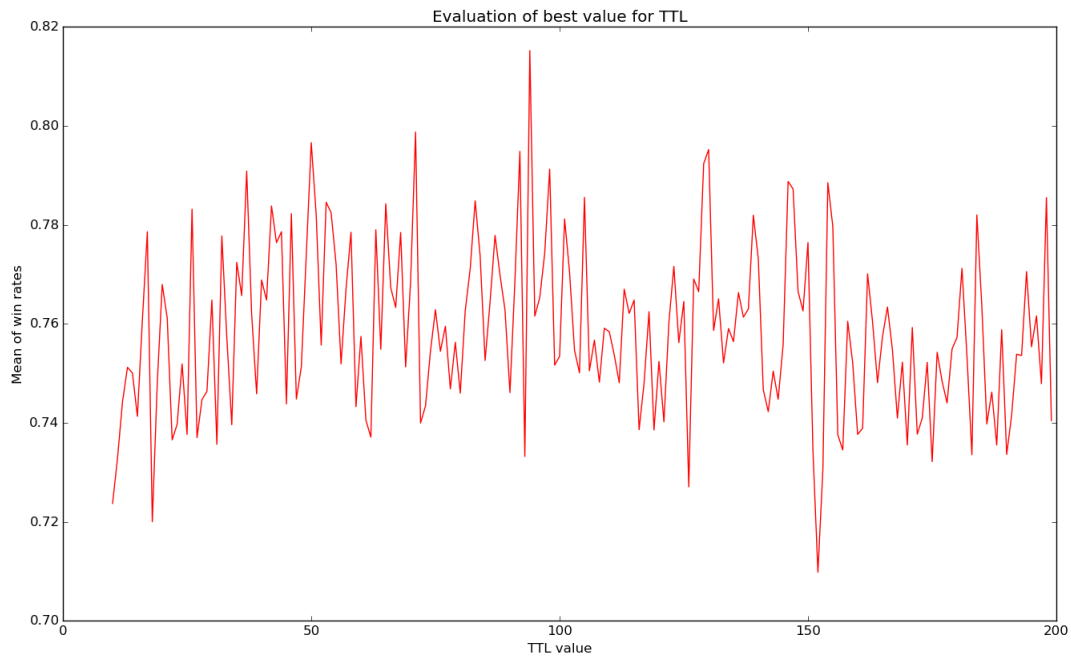


Figure 14.1: Parameter evaluation of TTL for Opponent Tree AI with time evaporation

player with better performance than the previous models. The drawback is that it achieves a little bit poorly performance against the *Positive random player* as compared to Section 12.6. And if we look at Figure 14.3, we also notice this model is efficient against the *Switch strategy player*. It has a smaller variance, and the gap between the green and the red lines is greater (as compared to the previous models).

This gain of performance is explained by 2 factors :

1. We only consider the TTL last games (TTL is equal to 94). This allows this model to adapt itself quickly against any player. This is especially useful to win against the *Switch strategy player*.
2. The latest games have a bigger impact than the first games. In other words, the moves played recently are considered as more likely to be played again in the current game. In this way, a change in the strategy of the player is more quickly spotted and countered.

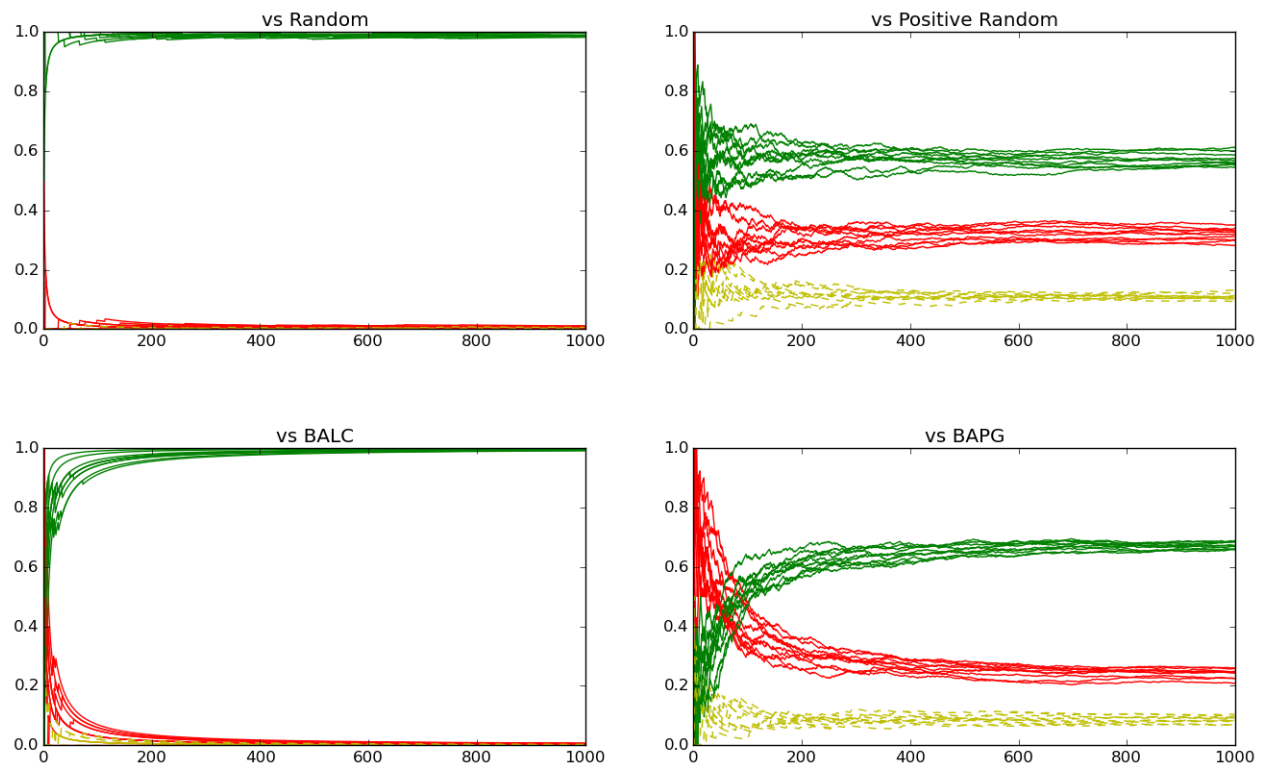


Figure 14.2: Opponent Tree AI with Time Evaporation

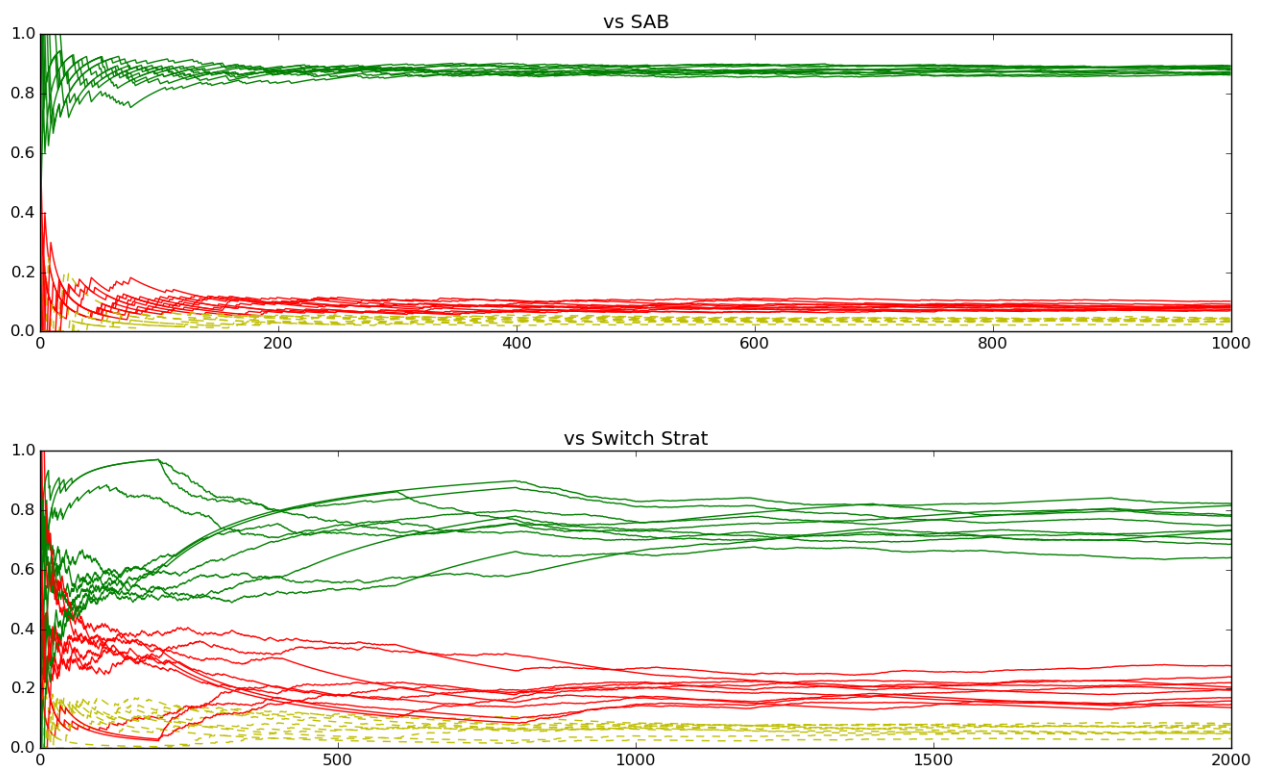


Figure 14.3: Opponent Tree AI with Time Evaporation

Chapter 15

Game Matcher AI

In this game, N-gram model can be used in two different ways : on the rounds and on the games. Previously, we only used this model on the rounds. The idea with this new model is to try the N-gram model on the previous game(s) to predict the next game.

15.1 Model

To make it simple, we have all the previous games in a database and we want to determine which game is most likely to be played now. To do so, with the N-gram model, we check the N-1 previous games and look into the database for the games with the same history (the N-1 games). To do that, we will have to look up into the database using SQL queries. Because finding two matching games is already very unlikely, we limit the history to one game : that is a bi-gram (N=2).

Even with a single game in the history, a full match is almost impossible when we don't have enough data. That's why we implement an "incremental" matching. In order to do it, we first retrieve all the games matching the first round of the history. For a round in the database (on the left of the *equalities* below) to match a round of the history (at the right of the *equalities* below), we require that :

1. $roundNbr = roundNbr_h$
2. $player1Choice = player1Choice_h$
3. $player1Bid = player1Bid_h$
4. $player2Choice = player2Choice_h$
5. $player2Bid = player2Bid_h$

If we find a match, we try to retrieve all the games matching the first round and the second round of the history. Every time we have a result, we retry to get a match by adding the next round of the history. If we have no result, we stop the search, and we keep the previous match (the one with a result).

Actually we are not interested in matching games but rather in the games following directly these games. Again this is done with a simple query to the database to get the next games. Once we have the games that are the most likely to be played after the history (we call them result games), we can perform probability counts on these games. To compute the probability of a move to be played at round i , we count the number of times this move appears in the result games at round i and divide it by the number of result games.

This produces the probabilities of the opponent to play certain moves. We use these probabilities as in the Opponent Tree AI (see Section 12). It means an AI's move that best counters an opponent's move will be played with the same probability as this opponent's move.

Let us show an example to make it clear. Let's assume that the previous game, the history, is the table shown on top of Figure 15.1. What the algorithm does is looking into the database for games that match the history. Let's assume that no game fully matches the history and that best matching games are games 12, 33 and 42 shown on the left below the history in Figure 15.1. These games match up to round 2. Once the algorithm has identified these games, it skips to the next games. These are the games 13, 34 and 43 (shown on the right in Figure 15.1).

In these games (13, 34, 43) we want to know which move is the most likely to be played at a certain round (the current round for the current game). For example at the first round, we observe that the opponent (P1 by convention) has played the following moves : (5,2), (5,4) and (5,5) (see Figure 15.1). Because they were all played only once they have the same probability $P[(5,2)] = P[(5,4)] = P[(5,5)] = \frac{1}{3}$. But for round 3, the opponent played (2,2) twice and (3,2) once. In this case, the probabilities are different : $P[(2,2)] = \frac{2}{3}$ and $P[(3,2)] = \frac{1}{3}$.

As you can see here the probabilities are not very diversified. But that's because we didn't have a lot of matching games. The more data we will have, the more matching games we will find and the more accurate the probabilities for this model will be.

15.2 Results

If we look at Figure 15.2, we notice this model has the same performance against the *Random player* and is just a bit less efficient against the *Positive random player* than the previous model. However, this model learns a little faster the *BALC player's* strategy. For the *BAPG player*, the AI takes more time for a poorer results.

Now if you look at the second figure (Figure 15.3), we also notice this model performs in a poorer manner than the previous model against the *SAB player*, but this model is worse against the *Switch strategy player*. Indeed, the variance is larger, and it globally has a lower win rate.

This performance is explained by the fact that, if a player has a fixed strategy, we will find the games that are similar to that strategy into the database and thus we will be able to counter them. For the *BAPG player*, since it plays "good" moves with randomness, this model has trouble to identify the next move that the opponent will play.

roundNbr	P1Choice	P1Bid	P2Choice	P2Bid
1	5	4	4	1
2	3	3	3	2
3	2	2	2	1
4	1	1	1	1

history

roundNbr	P1Choice	P1Bid	P2Choice	P2Bid
1	5	4	4	1
2	3	3	3	2
3	2	1	1	1

Game 12
Matches history
up to round 2

roundNbr	P1Choice	P1Bid	P2Choice	P2Bid
1	5	2	5	5
2	4	3	4	2
3	3	2	2	1
4	1	1	1	1

Game 13
After matching game

...

roundNbr	P1Choice	P1Bid	P2Choice	P2Bid
1	5	4	4	1
2	3	3	3	2
3	2	3	2	2
4	1	2	1	1

Game 33
Matches history
up to round 2

roundNbr	P1Choice	P1Bid	P2Choice	P2Bid
1	5	4	5	5
2	3	1	4	3
3	2	2	2	1
4	1	1	1	1

Game 34
After matching game

...

roundNbr	P1Choice	P1Bid	P2Choice	P2Bid
1	5	4	4	1
2	3	3	3	2
3	1	1	1	1
4	2	2	2	1

Game 42
Matches history
up to round 2

roundNbr	P1Choice	P1Bid	P2Choice	P2Bid
1	5	5	5	3
2	4	2	4	3
3	2	2	1	1

Game 43
After matching game

...

Figure 15.1: Database example for Game Matcher AI

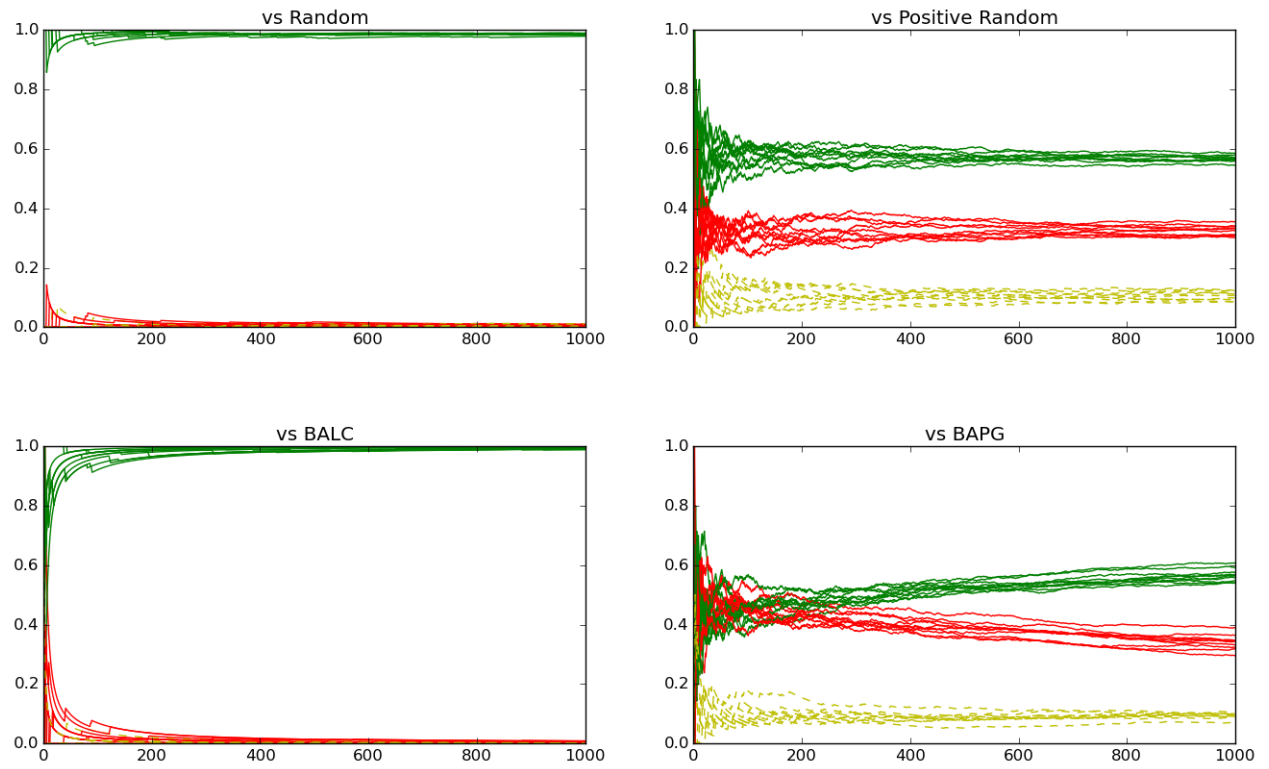


Figure 15.2: Game Matcher performance



Figure 15.3: Game Matcher performance

Chapter 16

N-gram N-gram AI

So far, we've used the N-gram model in only one way : on the rounds. But there is another way to it : on the games. The idea of the model is to combine the two ways of using the N-gram model. In other words, we will apply the N-gram on games, like in the Game Matcher AI (see Section 15.1), but we will also apply the N-gram on rounds like the opponent tree AI (see Section 12).

16.1 Model

The idea is to implement a bi-gram on games, and to expand the history thanks to the game we are currently playing. An example will be clearer : given the previous game we've played (we will call it our *history*), we can look into the database to discover which games are similar to that *history* in order to identify the games played after it. Imagine this is the lastest game we played :

RoundNbr	AI	Opponent
1	(5,1)	(4,1)
2	(3,1)	(2,1)
3	(1,2)	(1,1)

Table 16.1: History

Let's look into the database and assume we found two games following the *history*. These games are represented in Table 16.2. We found the games in the database more or less as described in the previous section (see Section 15.1). The difference here is that we only want a perfect match. Instead of having the games that match the best, we want the games that match totally, and if it's not possible, we have an empty result (see Section 16.1.1).

Now, we know what is the opponent's next move for the first round of the next game : (5,1). Given the result of the search in the database, the model assumes that only (5,1) is likely to be played by the opponent at this point. So, it will identify the move that wins against (5,1) with the gain function (see Equation 12.1.1) : it's (5,2) and it assigns to a 100% probability to be played because no other move were found with the search.

(a) Game 1			(b) Game 2		
RoundNbr	AI	Opponent	RoundNbr	AI	Opponent
1	(4,2)	(5,1)	1	(5,2)	(5,1)
2	(3,1)	(2,1)	2	(4,2)	(4,1)
3	(1,2)	(1,1)	3	(3,1)	(2,1)
			4	(1,2)	(1,1)

Table 16.2: Games following matching games

Now, imagine that the opponent effectively plays move (5,1), and we play move (5,2).

The first round is over, and we can add this information to our *history*. Now, the first game found in our database does not match anymore with our *history* (previous game + current game). We can remove it from our list of next potential games. That's the N-gram on rounds. We continue like this to predict the next opponent's moves.

16.1.1 If we find nothing in the database

The description of the algorithm looks pretty good, but only on paper. Because, actually, we have low probability to find enough perfect matches in the database in order to make good predictions. It is unlikely to find perfect matches, and that the games next to this perfect match, matches with the current game.

It's only because there are a lot of possible combinations of moves in a game that we need to have a huge database to find a complete match.

To solve this problem, this AI plays like the Opponent Tree AI if there is no match at some point (at the beginning or after expanding the history with the current game).

16.1.2 A good advantage

We can notice that this model and the previous one (see Section 15.1), are pretty accurate to predict the first round of a new game. As they both apply an N-gram on games, if there is a strategy, they directly find what will be the opponent's move at the first round of a the new game.

16.2 Results

This one has also good performance, but, globally, not as good as the Opponent Tree AI with time evaporation with list (see Section 14.2). If we look at Figure 16.1, we notice that this model doesn't succeed to beat the *Random player* perfectly (with a win rate of 1). This model also has a poorer win rate against the *Positive random player*, but we notice some learning as the number of games increases. It can completely beat the *BALC player* since the beginning because in the training it already played against this player and is able to find those games and deduce the player's strategy from them. Against the *BAPG player*, this AI does some learning of the strategy but it is even slower than in the previous model. Now, if we look at Figure 16.2, we

notice we have poorer performance against the *SAB player* and the *Switch strategy player* than the Opponent Tree AI with time evaporation with list, but it has still good performance.

This performance is explained by the same reason as for Game Matcher. The difference comes from the fact that this model is more accurate to spot matching games because it performs a full match and also takes the moves of the current game into account.

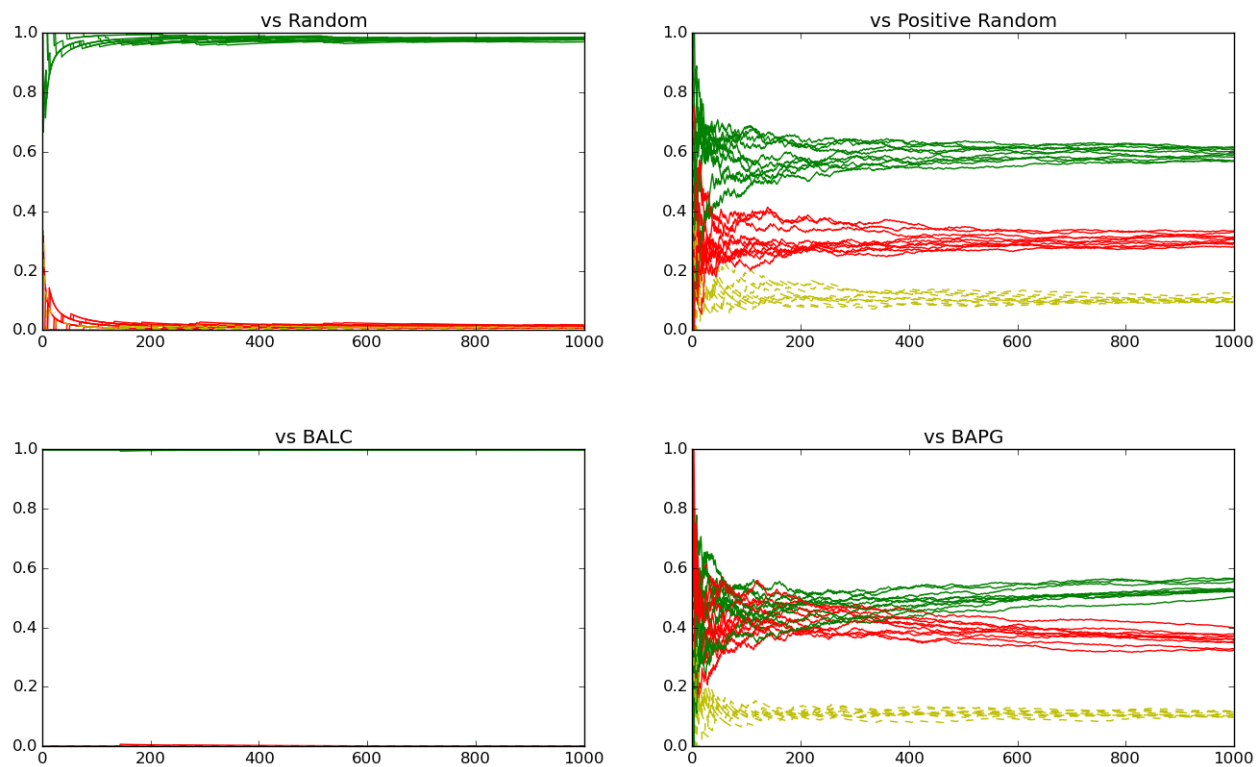


Figure 16.1: N-gram N-gram performance

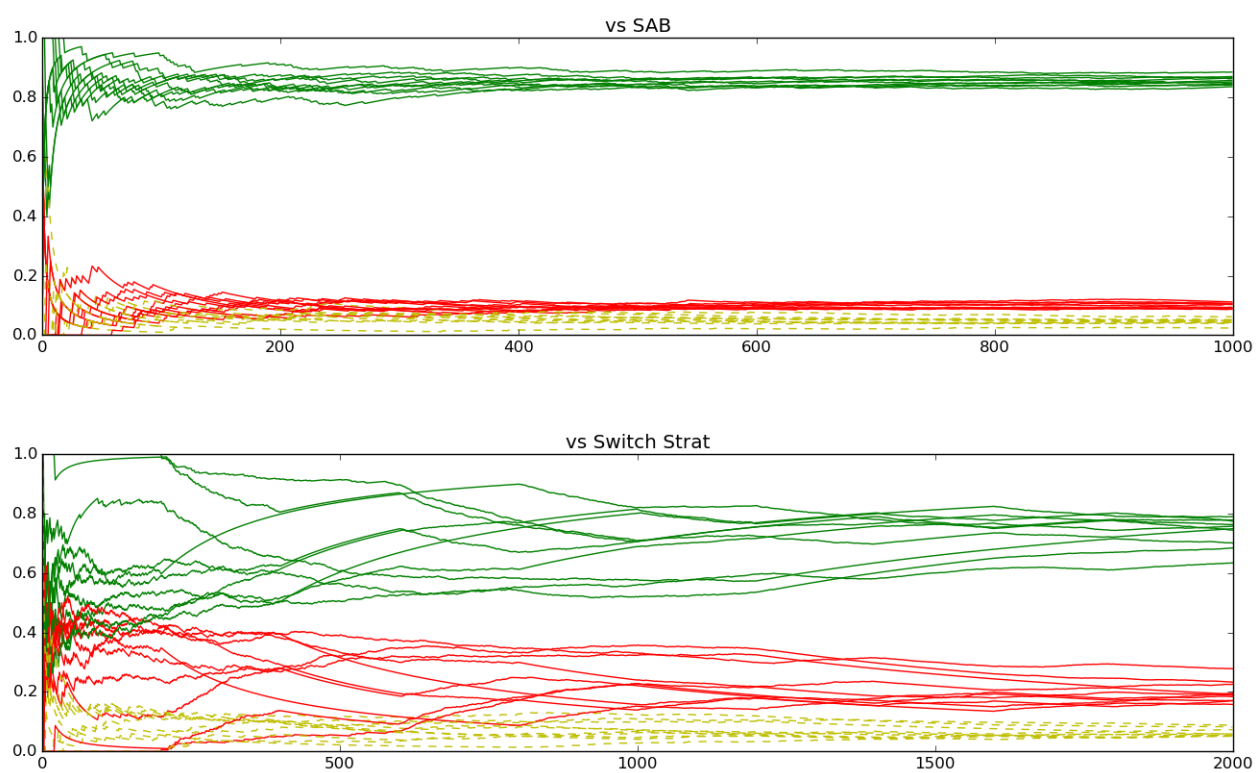


Figure 16.2: N-gram N-gram performance

Chapter 17

Conclusion

As for the first game, to simplify the conclusion, we summarised a lot of the information in Table 17.1. To put a fixed value, we took the arithmetic mean of the 10 green line (win rate) at the end of the graphs. The *Mean* row is the mean of the column; the numbers displayed in the *Best* column correspond to the number assigned to the best model against the specified player.

If we look at this table, we can observe that :

1. On average, the best AI is the Opponent tree with list with time evaporation with a mean of 0.8085.
2. On average, except for the N-gram based on wins, for Matrix AI and for the Opponent tree with win, the win rates of the other AI's are pretty close to 0.8085.
3. The best AI against most players is the Opponent tree with list with time evaporation.
4. Even if the Opponent tree with list with time evaporation is not the best against all players, it still has very good performance against them.
5. As soon as we have an AI with the gain function, we can easily win against the *Random player*. This is only because the AI will not play any obviously bad moves while the *Random player* will.

When the player doesn't play obviously bad moves, it becomes more difficult to beat, as we can observe with *Positive random player*. We observe that all the AI's have some problems to reach a better win rate than 0.6 against this player. This is still due to the randomness of this player. Because the AI cannot correctly predict this player's next move, the AI cannot adapt its best responses.

The *BALC player* has a "good" strategy since it tries to maximise its money. That's why this player wins against the poorer AIs such as (1), (2) and (6) (see Table 17.1). But it has a fixed strategy so that, as soon as the AI can learn the opponent's strategy, it has no problem to counter it and consequently the player can't win a game anymore.

The *BAPG player* also plays a "good" strategy but the randomness of its betting choices makes it more difficult to predict and thus to defeat. That's why even the best AI cannot win

more than 70% of the games (as we can see the best AI against this player is (7) with a win rate of 0.6734).

The *SAB player* is easily defeated by all the AIs, even if this player uses randomness. The problem here is that a *SAB player* chooses its actions randomly. By experience, we know that the highest actions are the ones that must be played as soon as possible in the game. But this player can play (1,1) at the first round, letting the AI gain the money by betting on the highest actions. Even Matrix AI (2) which has poor performance can win against this player with a win rate of 0.7.

But the player that will bring us more information when testing our AI's is, in our opinion, the *Switch strategy player*. Because we assume that human players will not follow a fixed strategy or if they do, they don't follow it for a long time, we require that the AIs must be able to adapt against a player that changes strategy. As Opponent Tree with evaporation (7) and N-gram N-gram (9) AIs both have the best win rates against this player, we conclude that these AIs are the best of all the ones we devised.

Moreover, we observe that the best AIs are able to do better than any of the players we have created. As we saw earlier, the most difficult issue the AIs will be faced with is randomness because that means that the N-gram cannot produce significant probabilities for predicting the opponent's actions. But in this game, adding randomness in the players does not necessarily make them better. And the good point of the AI's is that they have a higher win rate than the opponent's win rate for any player. This indicates that even with randomness, the AI is able to spot the good moves it has to play in order to win and that it is sufficient to win against anyone.

AI										
Player	N-gram based on wins (1)	Matrix AI (2)	Opp. Tree AI (3)	Opp. Tree AI with list (4)	Opp. tree with list with smoothing (5)	Opp. Tree with list with win (6)	Opp. Tree with list with time evaporation (7)	Game Matcher AI (8)	N-gram AI (9)	Best
Random	0.4549	0.9646	0.9757	0.9801	0.9779	0.6763	0.9873	0.9861	0.9797	7
Positive random	0.0341	0.4809	0.5889	0.6054	0.6059	0.1705	0.5712	0.5667	0.5909	4 and 5
BALC	0.009	0.3748	0.9437	0.9556	0.9548	0.2130	0.9949	0.9929	0.9998	9
BAPG	0.0135	0.254	0.5358	0.5385	0.5371	0.1733	0.6734	0.566	0.5328	7
SAB	x	0.7009	0.8445	0.8409	0.8505	x	0.8799	0.8295	0.8563	7
Switch strategy	x	0.4376	0.6859	0.7192	0.6948	x	0.7441	0.6834	0.7375	7
Mean	x	0.5355	0.7624	0.7733	0.7702	x	0.8085	0.7708	0.7828	7

Table 17.1: Summary of results for second game

Part III

Smartphone Application

Chapter 18

Application overview

In this section, we'll briefly introduce our smartphone application. Figure 18.1a shows what the application looks like when we launch it. As you can see both players start with 10 coins (below the "Confirm" button). The only thing that can be done at this step is pressing the "START" button.

Once the game has started, the AI selects its move and the player can also select one move, as illustrated in Figure 18.1b. To do so, the player can select one of the possible actions (buttons with 1 to 5 coins on the top). And he can also select the bet (default 1) by using the spinner next to the text "Bet". Once the player has selected its move (action and bet), he pressed the "Confirm" button.

Figure 18.1c displays the screen of the game at this stage. Both the AI and the player played (5, 1) so they both lost 1 coin, as we can see on the scores below the "Confirm" button.

The player can now select an action among 1, 2, 3 and 4. The game continues this way until no action is available or a player's money drops to 0. This is shown in Figure 18.1d. As the player has 6 coins and the AI has 3, the player has won the game, as the pop up says.

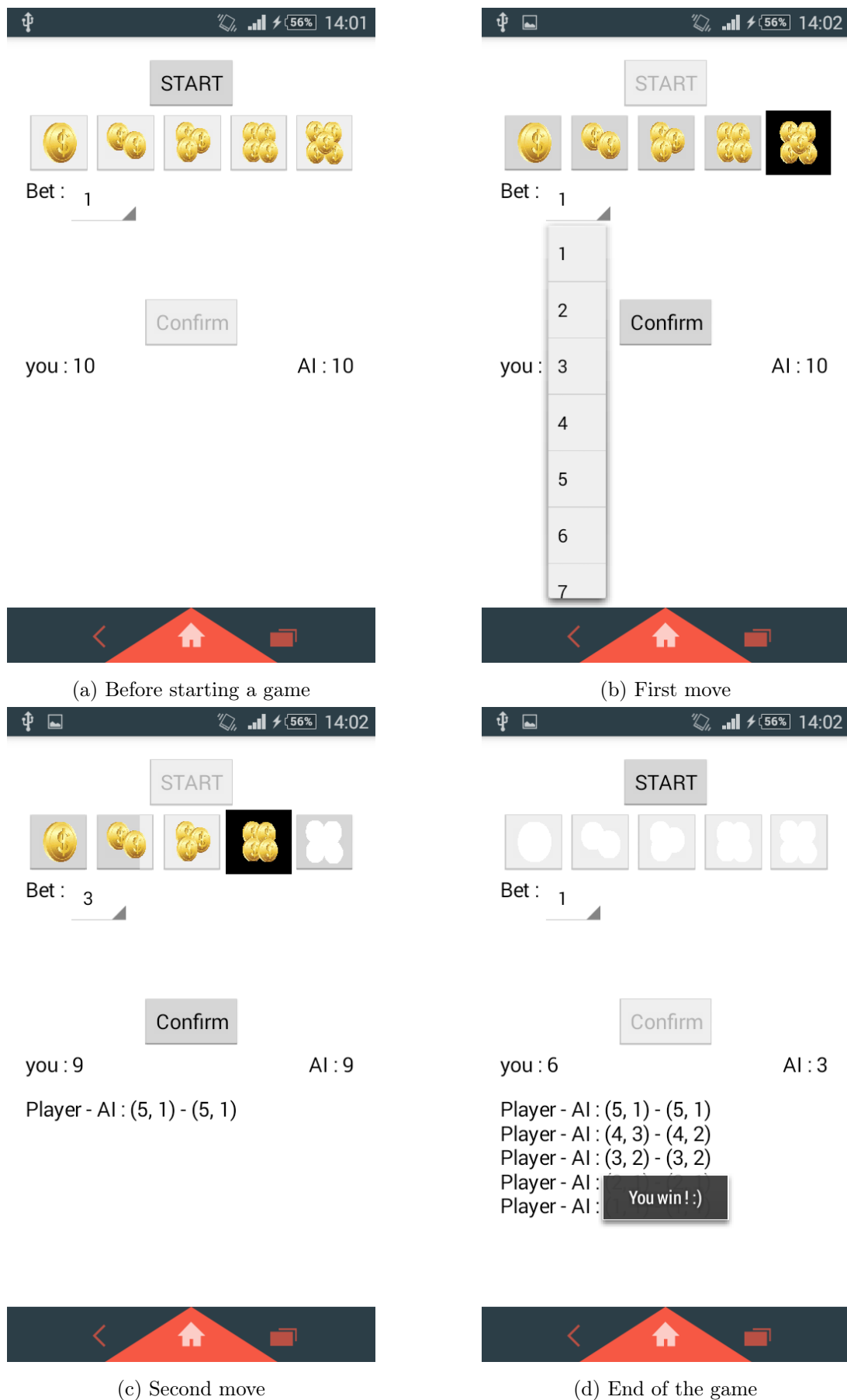


Figure 18.1: Game interface

Chapter 19

Smartphone limitations

19.1 Database limitations

19.1.1 Size limitation

As we use a SQLite database on Android, we are interested in the maximal size of the database. To manipulate our database on an Android device, we use the SQLiteOpenHelper class. This class is really well-built because it contains a method that allows us to know the maximum size it can reach. So we called this method and we found the following result : 1099511626752 Bytes. In other words, it means the database may grow up to $1099511626752B = 1073741823KB = 1048576MB = 1024GB = 1TB$. With such a size, we can assume the database on our smartphone will never reach its limit. The main size limitation would rather come from the memory size of the smartphone. The smartphone on which we tested the application has 2GB of memory. To check if it was an issue, we simulated 10000 games on the smartphone and checked the size of the database. For 10000 games, the size of the database is only 808KB. So the size of the database is not really an issue.

19.1.2 Access time

Gaining an access to the database on Android takes a non-negligible time. Actually, to running models like N-gram N-gram (Section 16) or Game matcher (Section 15.1) against humans is not a problem. Indeed, even if accessing the database is slow, humans are still slower.

However, we were still confronted with this problem when we drew our graphs. Indeed, when we drew our graphs, we made the models play against a player. In this case, the slow pace of the database was a problem. Making one graph took more than one day. To solve this problem, we had to run our models on a computer, and to change them slightly. Instead of updating the database continuously, we modelled the database with Java objects. It used up a lot of memory, but it was really faster.

19.2 Android limitations

We found only one limitation on the Android device, and it was already explained in Section 6.1.1. Otherwise, we weren't limited by our device.

We did our test on a Sony Xperia D2203. Although it is not an efficient device we've met no problems with it. So, we suppose that our models resource consumption is reasonable.

Conclusions

Thanks to both games, we saw that Machine Learning for games is possible on smartphone : we have no real limitations from the devices. However, as there isn't exist one model efficient for all games, Machine Learning has to be adapted according to the game. In our games, a good solution is to use a N-gram model to predict the opponent's moves. It seems that this model is well suited for this kind of games. Moreover, Machine Learning does not only depends on the game, but also on the opponent's predictability. As we saw in the first game, when the best strategy is playing randomly, if the opponent plays perfectly randomly, no model is able to win. But with the second game, we show how it is possible to implement a Machine Learning AI with significantly better performances than simple AI (i.e. the players against which we tested the models).

This thesis explains the process of creating a ML AI. The first thing to do is to analyse the game : what is the data and what will be subject to learning. Then, depending on the game and the data, one model should be selected among all the ones that can be found in the ML books. For the N-gram model, we explain an efficient way to implement it using trees. The next step is to improve the model using various techniques and data given by the game. For the N-gram model, we proposed different possible improvements and we compared their effects on the win rate of the model. It provides an overview of various approaches offered by the Machine Learning.

This thesis highlights one special thing : the gain brought by a Utility function to a model. It is conceivable to completely learn how to play a game by using ML (as we tried in Section 13) but if an Utility function is available, it is better to incorporate it directly into a ML model (as we did with the Gain function in Section 12). It also describes how it is possible to combine the two different approaches : utility based AI and Machine Learning.

Bibliography

- [1] S. Russel and P. Norvig (2014), *Artificial Intelligence : A Modern Approach*, Third Edition, Pearson.
- [2] I. Millington and J. Funge (2009), *Artificial Intelligence for games*, Second Edition, Morgan Kaufmann publishers.
- [3] Hal Daumé III (2012), *A Course in Machine Learning*, version 0.8 August 2012.
- [4] Nils J. Nilsson (2010), *The Quest for Artificial Intelligence: A History of Ideas and Achievements*, Web Version (<http://www.cambridge.org/us/0521122937>), Cambridge University Press.
- [5] Mark Watson (2002), *Practical Artificial Intelligence Programming in Java*, Version 0.84.
- [6] Gary Pollice, Stanley Selkow, George T. Heineman (2008), *Algorithms in a Nutshell (Chap. 7.6)*, O'Reilly Media, Inc.
- [7] T. Hastie, R. Tibshirani, J. Friedman (2008), *The Elements of Statistical Learning : Data Mining, Inference, and Prediction*, Second Edition, Springer.
- [8] "LINGI2263 - Computational Linguistics" course by P. Dupont and C. Fairon at UCL, <https://moodleucl.uclouvain.be/enrol/index.php?id=7865>
- [9] *Machine Learning in Games slides*, https://lectures.informatik.uni-freiburg.de/portal/download/75/6602/10_1_Games.pdf
- [10] *Merriam-webster's definition of AI*, <http://www.merriam-webster.com/dictionary/artificial%20intelligence>
- [11] *Wikipedia's definition of AI*, https://en.wikipedia.org/wiki/Artificial_intelligence#cite_note-Coining_of_the_term_AI-3
- [12] *Webopedia's definition of AI*, http://www.webopedia.com/TERM/A/artificial_intelligence.html
- [13] *Definition of ML*, <http://whatis.techtarget.com/definition/machine-learning>
- [14] *Definition of smoothing*, <https://pdfs.semanticscholar.org/945f/1c4efe98eb51a1943f60c0cd92df965cf8cf.pdf>

- [15] Dimps and Atari (2004), *Dragon Ball Z : Budokai 3*, PS2 game.

