

AI for games : Angry Birds

Dissertation presented by
Jérôme BERTAUX , Pieter HOLLEVOET

for obtaining the Master's degree in
Computer Science

Supervisor(s)
Pierre SCHAUS

Reader(s)
Yves DEVILLE, Cyrille DEJEMEPPE

Academic year 2015-2016

Acknowledgements

We would first like to thank our thesis adviser Professor Pierre Schaus from the Ecole Polytechnique de Louvain at Université catholique de Louvain. He steered us in the right the direction whenever he thought we needed it. He gave his time for us.

We would also like to acknowledge Professor Yves Deville and Cyrille Dejemeppe from the Ecole Polytechnique de Louvain at Université catholique de Louvain as readers of our thesis. Thanks for your time.

Authors.
Bertaux Jérôme
Hollevoet Pieter

Contents

1	Introduction	7
2	Angry Birds Basics	9
2.1	The materials	9
2.1.1	Stone	9
2.1.2	Wood	10
2.1.3	Ice	10
2.2	The Birds	11
2.2.1	Red bird	11
2.2.2	Blue bird	11
2.2.3	Yellow bird	12
2.2.4	Black bird	13
2.2.5	White bird	13
2.3	Other important game elements	14
2.3.1	Pigs	14
2.3.2	Slingshot	14
2.3.3	TNT	15
2.3.4	Summary	15
2.4	Naive Agent	16
2.5	Base framework	16
2.6	Documentation and samples of code	16
2.7	Benchmarks for agent comparison	22
3	State of the Art	25
3.1	Akbaba	25
3.1.1	Search	25
3.1.2	Simulation	26
3.1.3	Strategy	26
3.1.4	Competition results	26
3.2	AngryBER	27
3.2.1	Weighted Majority Algorithm	27
3.2.2	Naive Bayesian Network (NB)	27
3.2.3	Learning	28
3.2.4	Competition results	28
3.3	AngryHEX	29
3.3.1	The Tactic layer	30
3.3.2	The Strategy layer	30
3.3.3	Competition results	31
3.4	PlanA+	32
3.4.1	Strategies	32
3.4.2	Competition results	32

3.5	Datalab Birds	33
3.5.1	Strategies	33
3.5.2	Competition results	34
3.6	State of the art	35
4	First Agent	37
4.1	First Algorithm Idea	37
4.2	Preliminary improvements	37
4.2.1	Others possible improvements	40
4.3	Our heuristic	40
4.3.1	Heuristic Pseudo-code	41
4.3.2	Example	42
4.4	First agent pseudo-code	42
4.5	First results	43
4.6	Limitations	43
5	Second Agent with machine learning	45
5.1	Modification of the first agent	45
5.2	Machine learning module	47
5.2.1	Weka library	47
5.3	Naive Bayes classifier	48
5.3.1	Comparison with other classifiers	48
5.4	Attributes	51
5.5	Shot classification	52
5.6	Destruction threshold	53
5.6.1	Threshold : 10%	53
5.6.2	Threshold : 20%	54
5.6.3	Threshold : 15%	55
5.7	Overfitting	57
5.8	Agent pseudo-code	57
5.8.1	Learning part of the agent	58
5.8.2	Scoring part of the agent	60
5.9	Training set	60
5.10	Learning curve	61
5.11	Classification repartition for a set of shots	62
6	Comparing agents	63
6.1	Naive Agent VS our first agent	63
6.2	Our first agent VS our final agent	64
6.3	Our final agent VS the first three agents	64
7	Possible improvements	67
8	Conclusion	69
9	Bibliography	71
10	Annex	73

Chapter 1

Introduction

Angry Birds is a video game created by Rovio Entertainment in 2009. The game was immediately pretty successful and popular, 12 million copies of the original game had been sold in the first year. The goal is to destroy pigs protected by a structure composed of different building blocks that observe the laws of physics. The structure can be destroyed or displaced by shooting the angry birds at it. The fewer birds used and the more blocks destroyed, the higher the score will be.

Due to the nature of this game (large state and action spaces, various object properties, noisy object detection, not deterministic physical model), Angry Birds constitutes a really challenging task for AI.

The idea of this thesis comes from a competition, namely the **Angry Birds AI Competition** [1] which is based on the video game port on chrome. The competition has been organized each year since 2012 when the very first Angry Birds AI competition was held in Sydney, as part of the 25th Australian AI conference.

There is a framework for the agents developed for the competition, which includes Google Chrome plugins allowing our agent to interact with the game. Other available resources include a Naive Agent and the base framework as explained below.

The goal of our thesis is to develop an interesting agent that can play the game and score well relative to the benchmarks from previous years of the AIBirds competition.

We will start by a description of the Angry Birds game we consider, explaining the goals, all the tools we have at our disposal and the mechanics of the game. A state of the art section, explaining some of the most successful agents submitted for the competition follows. Then, we will discuss the first serious heuristic agent we created. We then follow up with the more complex machine learning based agent. Our agents are compared, evaluated and we explore a few improvement ideas in the subsequent sections.

Chapter 2

Angry Birds Basics

To understand Angry Birds, first we need to take a look at the different elements that constitute their world[2]. Screenshots giving a general idea of the 21 levels used in the competition benchmark are displayed in the Annex section on page 73. A copy of the benchmarks results available on the competition website on pages 23 and 24.

2.1 The materials



Figure 2.1: Example of Ice, Wood and Stone structures

2.1.1 Stone

Stone is the hardest breakable material present in the game that the Pigs use to fortify their constructions. It is extremely difficult to break for most birds, except for the Black Bird and the White Bird's eggs, this will be explained in more details in the following section about the birds page 11. However, they can be useful when fighting pigs, as the stones, if propelled on a pig, will usually kill the pig.

Only use weak birds on stone if they can make the structure topple rather than attempting to shatter it. If there is stone on top of a wood/ice structure, destroying the wood/ice directly underneath the piece of stone can cause major damage to the

structure because stone can destroy a large portion of the structure simply by falling on other blocks.

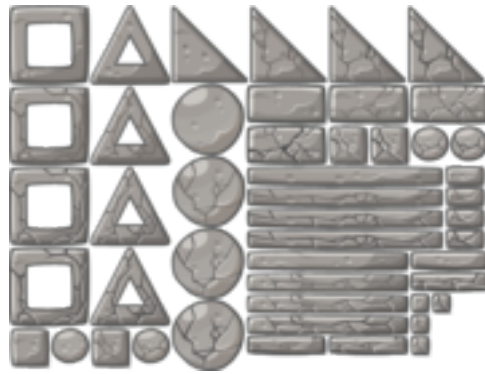


Figure 2.2: Stone blocks that can be present in the levels

As you can see in figure 2.2, stone blocks come in different shapes, but also various resistances. As the stone becomes weaker, more cracks appear, giving an idea of the damage already taken.

2.1.2 Wood

Wood is a type of block that has medium resistance. It is brown in color and best broken by Yellow Birds or Black Birds. Blue Birds (except when split) can only shatter one block of wood making them weaker than red birds when breaking wood.

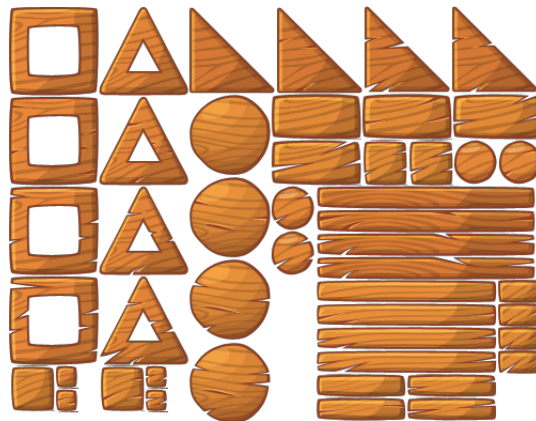


Figure 2.3: Wood blocks that can be present in the levels

Similarly to stone, wood also shows cracks when damaged.

2.1.3 Ice

Ice (sometimes called Glass) is light blue in color. Among the three kinds of material blocks, this one is the weakest and can be easily broken by all kind of birds. Levels with these blocks are often provided with Blue Birds, as they are very good at breaking these blocks.

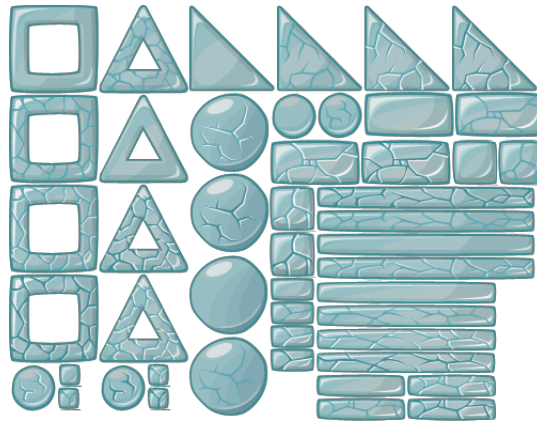


Figure 2.4: Ice blocks that can be present in the levels

Ice can crack like the other materials, but since it is so weak it usually only happens when the bird lost most of its energy which doesn't happen often.

2.2 The Birds

2.2.1 Red bird



Figure 2.5: Red bird

This is the basic bird, it is the only bird that has no active abilities, but is effective against everything.

Strength: Red birds have average strength on all materials and can shatter a single piece of ice or wood without problems.

Weakness: Red birds become weaker as the player encounters more difficult structures, that become more complex. As it is weak against stone, it can be more effective to make a stone structure tilt rather than breaking every piece of stone. It is possible to destroy stone but it can only be achieved by a red bird if the stone block is almost destroyed.

2.2.2 Blue bird



Figure 2.6: Blue bird

Strength: They ironically have a great strength against ice, as they can shatter a max of three thick walls of it (with an individual bird).

Weakness: Blue birds are the weakest of the birds and are useless against many structures. They don't cause a lot of damage to wood, and they are useless against stone, unless if they are used to topple it over so it capsizes a weaker structure.

They have an active ability that splits the bird in 3 when clicked during the trajectory of the shot. As shown in figure 2.7, the leftmost birds will disappear and be replaced by the three birds on the right. One of the birds will follow the same

trajectory as the initial bird, the two others will be sent above and under. It is important to use the ability to maximize the damage done by the otherwise weak blue bird.



Figure 2.7: Blue bird active ability illustration

2.2.3 Yellow bird



Figure 2.8: Yellow bird

Strength: Yellow birds are similar in strength to red birds, but a speed boost maximizes their strength making him more effective. He can shatter up to five walls of wood but only three without speed boost.

Weakness: Yellow bird is not as effective against square blocks. They don't cause a lot of damage to stone (even if sped up) for example, it would take three (two if sped up) birds to destroy a stone block. They can destroy ice, but their speed and strength will horribly decrease, making them less effective than blue birds versus ice. Most birds slow down when they hit ice, but Yellow Bird slows down after only breaking one bit of ice.

Yellow bird active ability is a speed boost, illustrated in figure 2.9. When activated, the bird's speed will drastically increase, with all the resulting consequences, the trajectory of the bird will be altered.

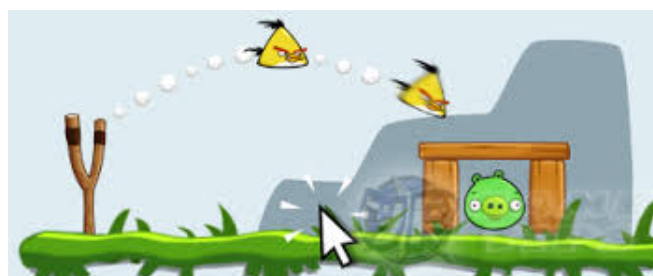


Figure 2.9: Yellow bird active ability illustration

2.2.4 Black bird



Figure 2.10: Black bird

Strength: Black birds are the strongest birds when facing stone, as they can shatter a maximum of four blocks before blowing up, and when he blows up, the explosion can destroy almost every material present in the small blast radius. Any material present near the explosion radius will be pushed and damaged. He is average on other materials.

Weakness: The black bird's ability is timed, making it harder for the player to get it in the right place for maximum destruction. It is odd how Black Bird can break through several pieces of stone but not wood.

The explosion ability will go off on its own, but the player can choose to instantly blow up the bird for a more precise explosion as shown in figure 2.11.

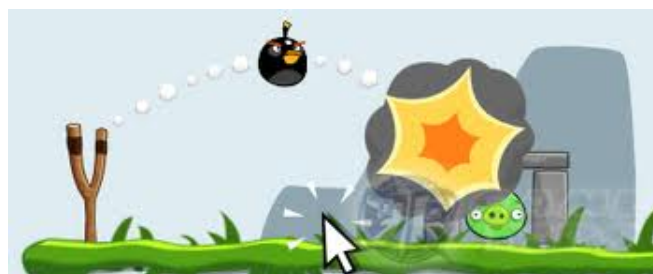


Figure 2.11: Black bird active ability illustration

2.2.5 White bird



Figure 2.12: White bird

Strength: The eggs she deploys deal great damage to ice and wood and medium damage to stone. The white bird itself can cause additional damage to structures after launching the egg.

Weakness: Although the egg deals good damage, it suffers from a small blast radius. Also, the bird itself before launching the egg deals poor (but not blue-poor as it can break 1 block of wood) damage.

The white bird's active ability makes it lay an egg that falls directly under the current position of the bird, propelling the bird upwards and making it almost impossible to predict the trajectory to effectively deal damage to structure. The bulk of the damage comes from the explosion of the egg itself.



Figure 2.13: White bird active ability illustration

2.3 Other important game elements

2.3.1 Pigs

The pigs are the main antagonists in the Angry Birds series. Pigs constantly steal the birds unhatched eggs, in the hope of cooking and eating them. In order to prevent the birds from taking back their eggs, the King Pig commands his army of pigs to construct many buildings and structures to protect them from the birds.

All of the pigs are light green in color, and come in several variations, see 2.14. First, there is the minion pig that comes in three sizes, small, medium, and large. Pigs will begin to wear different hats and/or accessories. Despite these hats being placed on the pigs, they can be hit without the hat being destroyed, or the hat can be destroyed protecting the pig from taking any damage.

Every pig knocked down in the game awards 5000 points, but slightly more if the accessory it is wearing is also destroyed in the process, if any is present. To clear a level all the pigs need to be destroyed.



Figure 2.14: Different kinds of pigs

2.3.2 Slingshot

The Slingshot is the main weapon used by the player to propel the birds on the pigs. The player can pull back on the slingshot strap, then release to fire the bird in the desired direction.

This game element is primordial for our agent since the approximate top of it will be used as starting point for the birds that are shot with the sling.



Figure 2.15: Slingshot

2.3.3 TNT

TNT Crates, also known as dynamite or TNT boxes, are an explosive used by Pigs in Angry Birds. Once struck, the TNT Crates explode, destroying objects within a short radius and send things flying away within a larger one. TNT Crates can prove to be quite useful for the player, since a large, fortified structure can fall when blown up.



Figure 2.16: TNT

2.3.4 Summary

The relative birds strengths are shown below. Note that even if a bird can't easily destroy a material, the target block can always be toppled. The fall can be more beneficial than a simple destruction.

Block	Strength	Birds that can destroy it effectively	Birds that can not easily destroy it
Ice	Weak	Red, Blue	Yellow, Black, White
Wood	Medium	Yellow	Blue, Back, White, Red
Stone	Strong	Black, White	Red, Blue, Blue

Table 2.1: Bird strengths

Another important element to consider is the fact that, when birds leave the screen they do not disappear. In fact, the bird will never leave the screen as it will zoom out to display it at all times. If a bird is shot in a high curve targeting the structure, he will always reach the targeted point. An example is shown in figure 2.17.

The levels are fixed, always have the same layout and the same birds to complete it. All the available birds to complete a level are known as soon as the level is loaded. The birds are given, the player cannot change the order the birds are shot in. Every bird not used when the level is cleared gives a 10000 bonus points.



Figure 2.17: Bird shot out of the screen still hit

2.4 Naive Agent

The website provides a very basic agent. This agent detects the slingshot and other objects, then shoots at randomly chosen pigs using either high or low trajectories. These trajectories are randomly chosen between high-arching shots (angle $\geq 45^\circ$) and direct shots (angle $< 45^\circ$) with a preference to direct shots. This agent uses predetermined tap-time intervals for both types of shots.

2.5 Base framework

The Base Framework consists of :

- The Angry Birds Extension works on the Google Chrome browser and allows to interact with the game window by offering some functionalities (Capture a screenshot of the game, execute actions).
- The Vision module : segments screenshot taken by the Angry Birds Extension, recognizes the bounding rectangles of main objects (birds, colors of birds, types of birds, pigs, slingshot, bricks and materials of bricks).
- The Trajectory Module : estimates the trajectory that a bird would follow given a target point.
- The Game Server : interacts with the Angry Birds Extension via a proxy module which can handle commands like CLICK, DRAG, MOUSEWHEEL, SCREENSHOT.

2.6 Documentation and samples of code

All information can be found in the ABDoc.pdf available on the competition website[1].

Minimum Bounding Rectangles (MBR) are detected by the vision module and include all the essential objects to model the level. MBRs are of one of the following types:

- Sling
- Red Bird
- Yellow Bird
- Blue Bird
- Black Bird
- White Bird
- Pig
- Ice
- Wood
- Stone
- TNT

An illustration of the MBR detection is given in figures 2.19 and 2.18. It can be seen that the four available red birds, the sling, the pig and the ice, wood and stone blocks are all detected.



Figure 2.18: Angry bird level



Figure 2.19: Example of MBR detection

The agent plays based on screenshots he gets from the game. In order for the different modules to work and allow the agent to interact, the game has to be set in low resolution. This also reduces the data to process at each step of the game.

Taking a screenshot of the state of game and analysing it is done by the following process :

```
// Start the game proxy
ActionRobot aRobot = new ActionRobot();
// Capture the screenshot
BufferedImage screenshot = aRobot.doScreenShot();
// Initialize the vision component
Vision vision = new Vision(screenshot);
// Get a list of MBRs of all the pigs in the screenshot
List<ABObject> pigsMBR = vision.findPigsMBR();
```

```
// Get a list of real shapes of all the pigs in the screenshot
List<ABObject> pigsReal = vision.findPigsRealShape();
// Get a list of MBRs of all the TNT on the screen shot
List<ABObject> tntMBR = vision.findTNTs();
// Get a list of real shapes of all the hills on the screen shot
List<ABObject> hills = vision.findHills();
// Get the type of the bird on the sling
ABType type = aRobot.getBirdTypeOnSling();
```

The first line create and initialize the game proxy, this proxy communicates with the game through plugins that have to be added to Google Chrome. The java object **ActionRobot** allows the agent to communicate the action he wishes to perform, including zooming in/out, shooting a bird, selecting the desired level etc.

The second line requests a screenshot of the game state from the java object **ActionRobot**. The following lines create and initialize the java object **Vision** that will decompose the screenshot. The **Vision** object allows to returns all the different blocks and pigs in the game. All objects in the game are represented by the java class **ABObject**.

There are many actions that the agent is allowed to perform, the most useful ones are represented below, in their java methods form :

- **cFastshoot(shot)** : Allows the agent to do a fast shot.
- **doScreenShot()** : Does a screenshot of the game and returns it in a **BufferedImage**
- **fullyZoomIn()** : Zooms in on the game window.
- **fullyZoomOut()** : Zooms out of the game window.
- **GoFromMainMenuToLevelSelection()** : These method goes from the main menu to the level selection window.
- **click()** : Clicks on the game window at the position we want.
- **cshoot(Shot shot)** : this method is the principal method to shoot a bird in the game. It shoots the bird and after that waits for 10 seconds.
- **drag()** : Drags the mouse from the position we want to an other position.
- **getBirdTypeOnSling()** : Returns the type of the bird on slingshot.
- **getScore()** : Gets the actual total score.
- **getState()** : Returns the state of the game.
- **loadLevel(int i)** : Loads the level given in parameters.
- **restartLevel()** : Restarts the actual level.
- **shoot(List<Shot> csc)** : Shoots all the level given in parameters and after waits for 15 seconds.

- `shootWithStateInfoReturned(List<Shot> csc)` : Shoots all the level given in parameters and returns the state of the game.

Each one of the in game objects detected on the screenshot is used to create an `ABObject` which extends `java.awt.Rectangle` and has the following attributes :

- `x` , `y` coordinate of the top left corner
- `width` : the width of the object
- `height` : the height of the object
- `type` : the object type (`"Hill"`, `"Sling"`, `"Red Bird"`, `"Blue Bird"`, `"Black Bird"`, `"White Bird"`, `"Pig"`, `"Ice"`, `"Wood"`, `"Stone"`, `"TNT"` }
- `hollow` : indicates whether an object has a hole in the middle or not

This is illustrated in figure 2.20. As you can see when the blocks are rotated, this representation causes complications as the point (x, y) isn't even a part of the block.

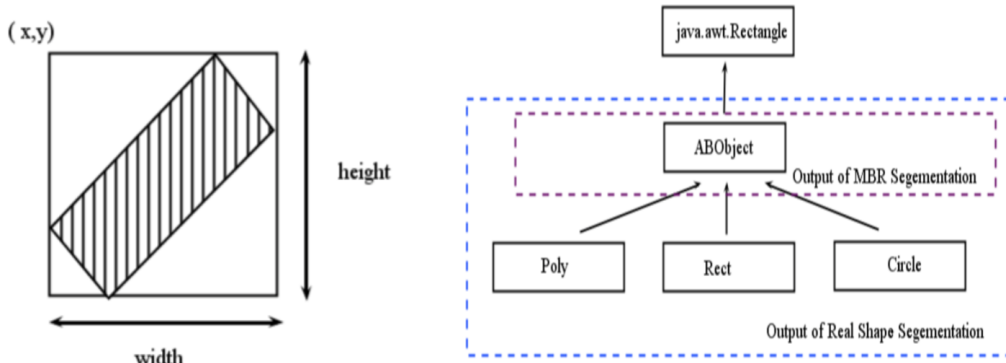


Figure 2.20: `ABObject`

The shapes of `ABObject` are classified into 3 groups (`Circle`, `Rect`, `Poly`). There is a java class for each group extended `ABObject`.

- `Circle` : represents a circular shape by the circle centre and radius.
- `Poly` : represents a polygon (`java.awt.Polygon`). *i.e* : triangles are represented by `Poly`
- `Rect` : represents a rectangle of an arbitrary angle. The `Rect` class introduce 3 more attributes : *pwidth*, *plength* and *angle*. The *pwidth*, *plength* maintains the length of the shortest and longest edge respectively. See figure 2.21 for an example.

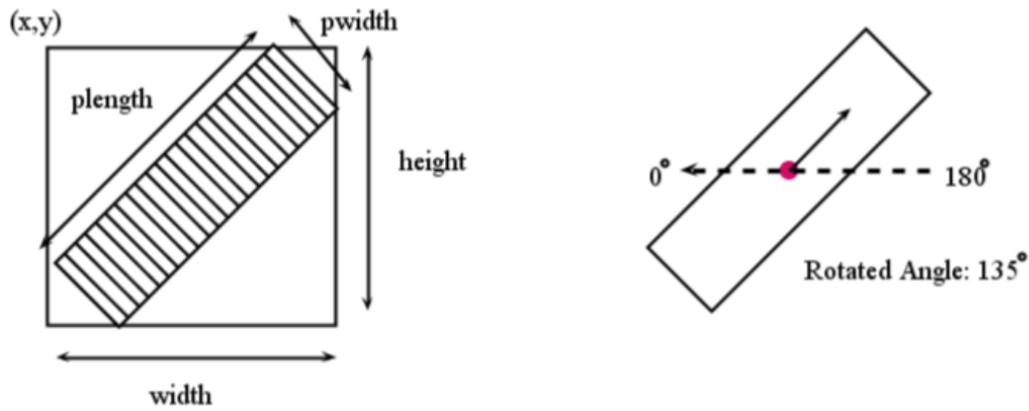


Figure 2.21: Rect's attributes

The possible game state allow the agent to know how well his shot was and help to plan next action. The states include:

- WON : a page shows the final grade and the stars you achieved
- LOST : a big pig tells you that you lost
- PLAYING : the game is not finished
- LEVEL SELECTION : the level selection page
- LOADING : the game is currently loading and a progress bar is showing on the page

The Trajectory Module is used to get the set of estimated release points given a target. This method requires the MBR of the slingshot and the target point.

```
// Initial a trajectory planner
TrajectoryPlanner tp = new TrajectoryPlanner();
// Get the estimated points
List<Point> pts = tp.estimateLaunchPoint(sling, target);
```

Finally some diverse methods :

```
// Return true if o1 supports o2
boolean b = ABUtil.isSupport(ABObject o1, ABObject o2);
// Return a list containing the subset of objects that support o2
// This method cannot deal with the support of angular rectangles
List<ABObject> objects = ABUtil.getSupporters(ABObject o2, List<ABObject>
    > objs);
// Return true if the target (point) is reachable by the shot
boolean b = ABUtil.isReachable(Vision vision, Point target, Shot shot);
```

The method `isSupport` returns a boolean. It is true if the first parameter supports the second one. That means that the first parameter is above the second one. In figure 2.22 the block B supports the block A.



Figure 2.22: `isSupport` : Example

The method `getSupporters` returns all the objects that support the block passed as parameter. For example in figure 2.22 the supporters of the block C are the block D and the block E.

The method `isReachable` returns true if the target point can be reached by the direct shot passed as parameter. For example in figure 2.22 the block E is not reachable for any shot because it is protected by other blocks. Remember the sling is always on the left and the birds will be shot to the right.

```
// Get the launch points
List<Point> launchPoints = tp.estimateLaunchPoint(slingshot, target);
//Choose a direct shot
releasePoint = launchPoints.get(1);
//Compute the tap time
tapTime = tp.getTapTime(slingshot, releasePoint, target, tapInterval);
// Get the coordinate to shoot
dx = (int) releasePoint.getX() - refPoint.x;
dy = (int) releasePoint.getY() - refPoint.y;
// Create the object shot
shot = new Shot(refPoint.x, refPoint.y, dx, dy, 0, tapTime);
//Shoot the shot
aRobot.cshoot(shot);
```

In the code example above, it can be seen how a shot is made. First, the trajectory module is used to estimate the possible trajectory from the slingshot to the target point. In this example, a direct shot was chosen. A high shot would have been chosen by selecting `launchPoints.get(0)`.

Then, the tap time needs to be computed according to the target point. This is done thanks to the trajectory module using the slingshot, the release point and the interval in percent of the trajectory. The next step is the computation of the coordinate of the release point according to the position of the slingshot.

Finally, the Java object `Shot` containing 6 parameters is created. The first four parameters specify a mouse track from `(refPoint.x, refPoint.y)` to `(refPoint.x + dx, refPoint.y + dy)`. The next parameter is the time to wait before shooting. The last parameter is the gap between the releasing time and the corresponding tapping time.

2.7 Benchmarks for agent comparison

The benchmarks used to compare the agents are the same as those used in the AI Birds competition. The levels are available in the annex page 73. We replicated the competition conditions, allowing a maximal time limit for the agent to complete the set of 21 Angry Birds levels, which is set to be a minimum of 3 minutes per level on average. The levels must be cleared automatically and without any human intervention. All levels of the current round can be accessed at any time. The agent's score is the sum of the best score obtained on each level.

For the evaluation of our agents, we simply let them clear all the levels one after the other and checked the time didn't exceed the limit, we didn't build an additional strategic module to utilize the leftover time. Other agents utilize this time to retry levels they failed or to try and improve level they already cleared.

The score for each level is the points the agent obtained, the total benchmark score is the sum of the highest score the agent obtained for each of the 21 levels within the allowed time. The tables 2.2 and 2.3 show the benchmarks available on the website competition [1]. The scores have unfortunately not been updated since 2014. We inserted the scores of our two agents.

There are various way to obtain points in the game :

- A pig kills gives 5 000 points.
- Destroying a block of any type is worth 500 points.
- An unused bird when the level is cleared is 10 000 points.
- The collision between a bird and a block gives some points depending on the amount of damage inflicted to the block.

Table 2.2: Benchmarks part 1

Rank	Team	Total Score	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7	Level 8	Level 9	Level 10
1	PlanA+	1002380	30480	62370	40620	29000	69440	36970	32020	47320	26440	56830
2	DataLab Birds	981120	31620	52000	41890	19790	70320	15700	45720	43190	50420	56790
3	AngryHex (2013)	974670	30390	54160	41890	28000	64440	24990	36900	24860	49570	50570
4	UCL Agent ML	964020	30760	61410	41900	18770	49580	25530	30050	57890	44900	72870
5	WISC	963160	29030	52180	41950	27850	65810	24640	25520	55810	33680	39170
6	AngryHex (2014)	960320	32660	52580	41910	19690	68090	25180	24680	43330	42740	54890
7	Angry Concepts	954030	29410	52250	41320	28160	64160	15660	24630	47170	48670	48290
8	Beau Rivage	952390	29760	43160	40500	28680	61000	33540	45780	49150	33110	42200
9	HungryBirds	951440	31210	53860	42040	20980	65700	28430	40580	27020	50710	53070
10	AngryBER	935330	28510	43420	41910	19390	62880	35610	31980	45050	48670	51980
11	MIT RedBacks	933120	32220	53860	30480	19380	70350	33470	45740	29180	33320	59190
12	Luabab	894840	29600	52180	40760	28030	66100	16860	32180	48330	41840	54110
13	Dan	893370	29210	42760	41610	27990	63840	25700	45990	23390	45930	49570
14	UCL Agent	892180	30920	34200	41980	27930	56100	25620	22800	38140	36380	58030
15	IHSEV (2014)	891590	29520	53070	41440	21860	56470	33470	29920	36630	40620	32300
16	Impact Vactor	886990	29710	60480	42040	19700	65440	33870	36880	25720	34770	46870
17	Black Forest Cuckoos	874650	29880	52390	41600	28980	65380	0	32380	56310	23050	52360
18	IHSEV (2013)	861830	29360	52780	41240	36810	51860	33790	30130	57570	24310	29860
19	Naïve Agent (2013)	858730	29510	52230	40620	20680	55160	16070	21590	25730	35490	32600
20	Sniper	785520	29760	42720	40500	18620	65850	35490	28780	48170	26960	35540
21	Wanderer	774190	30840	60400	41900	36770	63550	26910	22070	57780	51480	68740
22	Naïve Agent (2014)	756680	29760	43250	40180	10590	62490	14980	22150	35840	36050	52570
23	s-birds	754190	30080	26240	42240	19050	38160	36180	49120	38340	41670	54940
24	S-birds Avengers	706110	28520	51770	35380	27950	60920	26000	0	48010	49050	0
25	Lambdaers	705010	30970	52390	42070	19580	63750	26400	45880	57600	49370	61820
26	A.Wang	680640	22400	43320	31910	28080	48440	24770	23960	27580	42590	57610
27	Akbaba	641080	28640	35540	41910	19120	47470	0	23880	45380	36960	56000
28	AngryDragons	620390	28120	33440	30430	28820	65650	0	0	29340	25590	0
29	ObjectS	569000	29670	43200	40620	11720	65850	15570	21310	37170	22610	51250
30	AngryPKU	476100	18420	33280	24070	10120	35650	26450	0	27840	24660	30160
31	JulyPlayer	352340	29360	52520	35360	20690	46300	17400	21810	45370	48670	34860
32	FEI2	211870	28360	34160	30470	27650	64710	26520	0	0	0	0

Table 2.3: Benchmarks part 2

Rank	Team	Level 11	Level 12	Level 13	Level 14	Level 15	Level 16	Level 17	Level 18	Level 19	Level 20	Level 21
1	PlanA+	47240	58210	34010	65640	54910	57530	51190	52120	39440	45980	64620
2	DataLab Birds	50650	53420	32010	55640	46450	57380	48570	45730	35470	54680	73680
3	AngryHex (2013)	53510	57750	42010	58190	46550	63430	46820	50020	38460	46970	65190
	UCL Agent ML	47760	38970	47580	57010	49630	58800	47870	44460	38480	36100	64300
4	WISC	50000	54980	32450	65060	49200	63430	54750	44740	37370	47980	67560
5	AngryHex (2014)	53570	54860	41200	57150	41100	61470	50260	48050	36780	39010	71120
6	Angry Concepts	56130	58600	50360	58050	43340	55000	45990	44030	37880	43730	61200
7	Beau Rivage	57420	55600	35190	62770	47270	55890	49880	38820	29780	37020	75870
8	HungryBirds	56260	55410	30130	59780	41040	55310	43530	46770	32690	56050	60870
9	AngryBER	51550	54720	42510	45640	47090	50000	47740	44110	36450	35990	70130
10	RMIT RedBacks	45250	56910	42300	45640	41540	66570	39370	48530	30460	40760	68600
11	Luabab	45580	53040	41110	65640	28910	65670	39550	37680	16820	36970	53880
12	Dan	38570	54990	32270	57550	47280	63000	42770	48290	22040	36910	53710
	UCL Agent	40760	47450	34190	66650	35620	62970	43850	40710	34040	41640	72200
13	IHSEV (2014)	31570	61070	41330	59250	34830	54160	47830	42840	40100	40470	62840
14	Impact Vactor	57490	55230	49620	0	45860	58870	47070	48210	25040	37650	66470
15	Black Forest Cuckoos	54360	57750	29770	57560	55300	59940	52920	41410	36940	46370	0
16	IHSEV (2013)	59070	40960	42840	65640	30700	51730	47940	47320	30940	0	56980
17	Naïve Agent (2013)	46760	54070	49470	50590	46430	55210	48140	49430	37920	36790	54240
18	Sniper	51150	49190	26180	65640	0	56530	37980	40460	30390	55610	0
19	Wanderer	30620	42320	20050	58200	25520	38960	37560	35220	25300	0	0
20	Naïve Agent (2014)	39310	48660	30000	45640	44190	52300	39530	39590	29460	40140	0
21	s-birds	57950	47460	22550	63560	42990	65750	38790	39120	0	0	0
22	S-birds Avengers	48320	45150	22130	53760	41240	0	45000	45750	38200	38960	0
23	Lambdaers	44120	38390	29350	54340	44040	0	44940	0	0	0	0
24	A.Wang	47410	46980	27490	65640	36160	66550	39750	0	0	0	0
25	Akhaba	48460	52310	0	59420	0	47230	44260	54500	0	0	0
26	AngryDragons	39580	51340	30840	55640	33920	53800	45660	38390	29830	0	0
27	Objects	57930	45200	26570	65640	34690	0	0	0	0	0	0
28	AngryPKU	31740	45870	0	45640	32790	55890	0	0	33520	0	0
29	JulyPlayer	0	0	0	0	0	0	0	0	0	0	0
30	FEI2	0	0	0	0	0	0	0	0	0	0	0

Chapter 3

State of the Art

The agents of some of the previous winner of the Angry Birds AI Competition[1] are public and shared on the web site of the contest ¹. All the agents participating the contest also have to submit a short summary of their agent's functioning. Unfortunately, for most agents we cannot find much information except for their score. Some of the best agents will be described in this section.

The benchmarks have last been updated in 2014 so last contests could not be taken into account. A copy of the benchmarks is available at the end of the previous section (pp. 23 and 24) or on the website of the contest [1].

3.1 Akbaba

The idea of this agent created in 2013 is to simulate the physics of the game in order to make the best possible shot. They use the **Box2D**, an open source C++ engine, which is allegedly used by the Angry Birds game on chrome itself. Box2D includes collision detection, continuous physics with time and impact solver and other that are used to easily simulate bird shots. The idea is thus to mimic the physics by learning the in game parameters[3].

First, different reachable target positions in the scene are considered. Then, different possibilities to reach that target are evaluated in terms of incoming angle and speed of the bird.

3.1.1 Search

To make the search faster, the parameters of a shot are determined in a hierarchical fashion. The search space is split into smaller parts in every direction and one element is tested for each part. The outcome is estimated for each element by simulation. Then, only the parts that yielded the best results are looked at further.

In order to implement this hierarchical approach, qualitative (spatial) abstractions were used. **Quadtrees** were used for the subdivision of the search space in order to find a target point. They are a tree data structure in which each node has exactly four children. This allows to subdivide recursively a two dimensional space into four quadrants as shown in figure 3.1. During the search for possible target point, possible hit points are first generated along the edges of objects. Then, the scene is divided with those points using quadtrees.

¹<https://aibirds.org/>

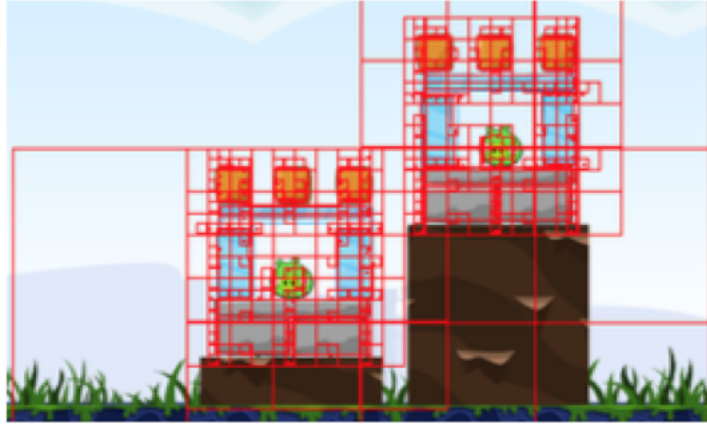


Figure 3.1: Scene subdivide with Quadtree

They test the nodes at each level of detail by running a simulation with the center of that node as a target point. The same method is applied to find appropriate parameters (the moment they activate the special ability of the bird, angle of the trajectory) for the trajectory to use to hit the target point.

3.1.2 Simulation

With 2DBox the agent tries different shots with different parameters to find those that maximize the estimated score. It uses a combination of heuristics that tries to replicate the score achieved in the actual Angry Birds game by the corresponding shot.

The quality of their agent depends on the accuracy of their simulation. They need to mimic the physics of the game as closely as possible and try to improve the quality of the agent which strongly depends on the learnt parameters and the quality of the scene parameters gathered by the vision module.

3.1.3 Strategy

They play all level once. Then they retry levels that the agent failed in the first try. But for the second try they broaden some of parameters to increase the likelihood of finding a solution.

3.1.4 Competition results

Their agent didn't perform well, finishing at the 25th place as can be seen in the benchmarks (pp.23 and 24). Their agent didn't even beat the naive agent that shoots at random pigs, allegedly the random agent performed well that year.

3.2 AngryBER

The team of the University of Wisconsin-Madison has decided to create an agent for the competition in 2014. The key feature is a powerful inference mechanism. In order to achieve this, they used an efficient tree structure to encode and represent the game state obtained through screenshots. This approach consists in learning the general task of playing Angry Birds rather than learning how to play specific levels. A key aspect of that design is that the features provided to the learning algorithms are a function of the local neighborhood of a shot's expected impact point[4].

To collect training examples they used the Naive Agent provided by the contest website.

3.2.1 Weighted Majority Algorithm

The **Weighted Majority Algorithm** (WMA) is a machine learning technique that uses a pool of prediction algorithms to build an efficient compound algorithm. The algorithm evaluates examples by taking a weighted vote among the members of this pool. It learns by altering the hypotheses associated weights when it makes mistakes. An advantage of this algorithm is that it can operate without any prior knowledge about the quality of the prediction algorithms in the pool.

They chose the WMA because it is able to handle inconsistent training data, is simple to implement and runs very quickly.

In their implementation, each prediction algorithm is simply a boolean feature or its negation describing a property of the localized representation of the game state. Each feature has a weighted vote *for* and *against* a shot. Each shot contains a net score that is the difference between the sum of all the features of weighted votes *for* and the weighted votes *against*. Votes that lead to incorrect decisions have their weights reduced.

3.2.2 Naive Bayesian Network (NB)

The Naive Bayes (NB) classifier encodes a directed, acyclic graph with conditional independence relations among variables. In Naive Bayes, a dependent class variable Y is the root and feature variables X_1 through X_n are conditioned by Y 's value. The naive assumption is that each feature X_i is conditionally independent of every other feature X_j for $j \neq i$ given Y .

The goal is to estimate the probability $p(Y|X_1, \dots, X_n)$. For a given game state the Y is *goodshot* and the X 's are the features used to describe the game's state and the shot angle.

They take the ratio $p(Y|X_1, \dots, X_n)/p(\neg Y|X_1, \dots, X_n)$ that represents the odds of a favorable outcome given the features of the current state. For each feature they compute the probability for *goodShot* and *badShot* and the ratio of these probabilities is the score of a candidate shot.

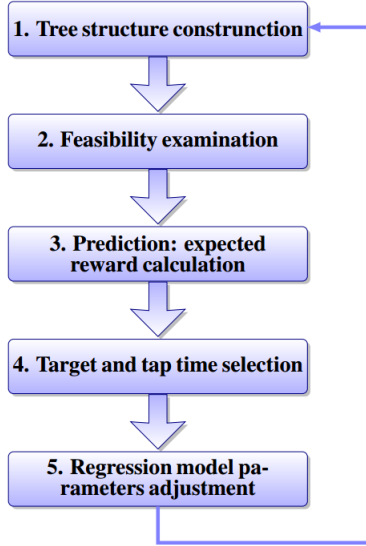


Figure 3.2: AngryBER proposed method

As shown in figure 3.2, their agent repeats 5 steps. First, the level is modeled using the tree structure. Then, node reachability is evaluated with an estimate of the expected reward. Target and tap time selection takes into account the target and the type of bird to evaluate the optimal time to activate the bird's power. Finally, the parameters of the regressor are adjusted using a recursive estimation framework.

3.2.3 Learning

The learning algorithm is trained on a set of examples generated by simple agents with varying degrees of intelligence. They are categorized into positively and negatively labeled datasets. They implemented agents that learn how to play each specific level to get enough positive data (winning games). After training and tuning on the examples they use their algorithms to play Angry Birds by scoring a set of a dozen candidate shot angles, choosing one of the highest-scoring candidates. The probability of selecting a shot is proportional to its score.

3.2.4 Competition results

They placed 9th on the competition in 2014 (benchmarks pp.23 and 24), they concluded that the main drawback of their approach is to find the appropriate number of basic functions. During training, some of their coefficients vanish due to priors, only a select number of coefficients are considered relevant due to the training data that might not be the best reflection of actual levels.

3.3 AngryHEX

The agent is a joint work project between the University of Calabria (UniCal) and the Vienna University of Technology (TU Vienna). They participated in the contest in 2013, 2014 and did pretty well each time. Interestingly their agent from 2013 performed better than the one of 2014.

It is based on declarative knowledge bases, and features a combination of traditional imperative programming and declarative programming that allows to achieve high flexibility in strategy design and knowledge modeling, dealing well with incomplete knowledge. In particular, it relies on the use of Answer Set Programming (ASP) and HEX programs. Logic programming is used for decision tasks such as which target to hit at each shot (tactic gameplay) or which level to play at each turn, and which way (strategic gameplay).

Answer Set Programming (ASP) is a form of declarative programming oriented towards difficult search problems. It is based on the stable model (or answer set) semantics of logic programming. In ASP, search problems are reduced to computing stable models, used to define a declarative semantics for logic programs with negation as failure. Answer set solvers, are programs used for generating stable models, which are used to perform the search. The computational process employed in the design of many answer set solvers is an enhancement of the complete total backtracking algorithm and, in principle, will always terminate.

HEX is a multi-paradigm programming language that supports both static and dynamic types, and was designed with the core principles of simplicity, readability, versatility and scalability to allow developers to create a diversity of types of computer programs with modern language features, succinct syntax and semantics that are built into the core of the language construct. HEX is an extension of ASP that is well suited for external knowledge and external sources of computation must be integrated in the same database.

The agent is based on the **DLVHEX** system, developed at TU Vienna and is a logic-programming reasoner for computing the models of so-called **HEX-programs**.

A **DLVHEX** solver is used to communicate with an *Executor* that they add to the base framework. Executor encodes the information about the current state of game into logic assertions and runs **DLVHEX** on the composed HEX-program and get the output so identify the next target point. Agent functioning is shown in figure 3.3.

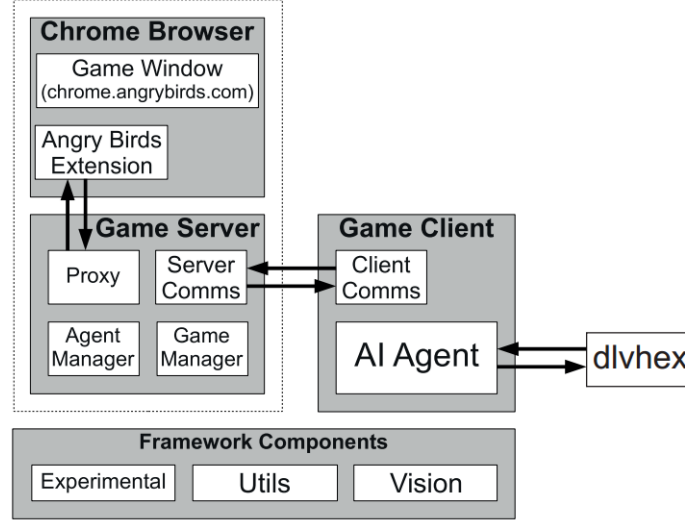


Figure 3.3: AngryHEX base architecture

They made improvements to the base framework. They improved the Vision Module adding the possibility of recognizing the orientation of the blocks because the base framework doesn't recognize blocks that are not parallel to the coordinate axes. They also improved the Trajectory Module with the possibility of aiming a trajectory at the left-top side of objects because the base framework aims only at the center of the object.

They developed 2 different intelligence layers for their **AngryHEX** agent :

3.3.1 The Tactic layer

The tactical layer plans shots, and decides how to complete a level. It is implemented using the **DLVHEX** solver that computes optimal shots based on information about the screenshot and on knowledge modeled within the HEX program P_{AI} . It works like this :

- Input : Scene information encoded as a set of logic assertions S (coordinates, size, orientation of pigs, materials of bricks, slingshot) and a knowledge base P_{AI} encoding knowledge about the game play.
- Output : The answer sets of $P_{AI} \cup S$ that describe the target point to aim and information about the shot.
- Other information are accessed within P_{AI} like physics simulation, if an object can be pushed by another, etc.

3.3.2 The Strategy layer

This layer decides the order of levels to play and possible multiple attempts to the same level. It keeps tracks of previously achieved scores and previously target objects. It is implemented according this scheme :

1. The agent plays each level once.
2. Levels where the agent's score is too low with respect to the current best score are replayed.

3. Levels where the Tactic layer achieved a score higher than the current best scores and that have the minimum difference from the best score are replayed.
4. The Strategy layer play a random level.

For AngryHEX the tactical layer is crucial, since it is prompted for each shot it need to optimized for the agent to stay within the time limit. The size of the knowledge base and the size of the current scene will have an impact on the performance of the tactical layer.

3.3.3 Competition results

The agent performed very well, finishing 3rd with one of the best score on the known levels (benchmarks pp.23 and 24), but scored poorly on the hidden levels, we unfortunately don't have access or insight for those levels.

3.4 PlanA+

The Department of Computer Engineering of Sejong University created an agent for the competition in 2014. The agent won the competition and is currently positioned first in the benchmarks (pp.23 and 24) available on the competition website. Unfortunately, they didn't provide much information about their agent compared to other participants[5].

3.4.1 Strategies

The core of their agent uses two heuristics, at the first run of each stage, the agent always uses the first strategy, if the level is not cleared, the second will be strategy.

First strategy considers each pig and, if present, the TNTs, and calculates the outcome of each shot based on the approximation of the objects it has to destroy to reach the wanted target. They put a hard limit of two trajectories for each target. Since different strategies have to pass different objects, the scores have to be normalized.

The second strategy is similar to the first, in the sense it selects the same potential targets, but it number of pixels crossed by the bird are used for scoring.

The tuning of the parameters of the two strategies was done manually at first, and then a greedy optimisation was applied.

They concluded that the trickiest part of the problem was that it is impossible to speed up the scoring process, they ran multiple computers in parallel to speed it up.

3.4.2 Competition results

As already stated the agent finished first on the benchmarks (pp.23 and 24), with the highest score ever obtained.

3.5 Datalab Birds

DataLab Birds from the Czech Technical University in Prague participated for the first time in 2014. Since `Plan A+` didn't provide documentation it's also the best performing documented agent we have [6].

3.5.1 Strategies

The success of a given strategy strongly depends on the layout of the level we try to beat. Their main idea was to find a small set of strategies that could efficiently cover all the levels. The core of their agent is thus planning. The goal is to select the best strategy to play given the block configuration, the birds, and the reachable targets with associated trajectories.

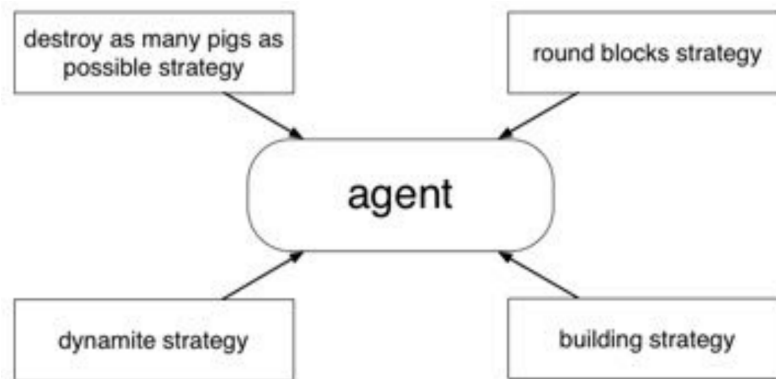


Figure 3.4: Datalab agent strategies

Dynamite strategy

As you probably have guessed, the strategy consists in targeting TNT, but since hitting the TNT for the sake of hitting it is not really useful, there need to be additional conditions. The position of the TNT has to be taken into account, and more precisely the distance between the TNT and the closest pigs, because obviously pigs will take more damage the closer they are from the explosion. Dynamite is also more potent when there are a lot of stones nearby, the stones can be propelled by the explosion and hit pigs or structures that are further away.

Building strategy

This strategy aims to find structures near pigs, and is designed to solve cases where pigs are shielded by constructions and unreachable by direct hits. When a pig is protected by a structure, a direct hit on the pig may cause the structure to transform into an even more potent protection for the pig. It is thus important to only take the shot if the outcome is beneficial.

They defined criterion to select suitable target blocks, based on the material, the bird, the position and orientation of the block.

And finally additional special cases are used if the structure is a Pyramid, a Skyscraper or a Rectangle, where the ideal aiming targets are very different. Example structures are given below in figure 3.5.

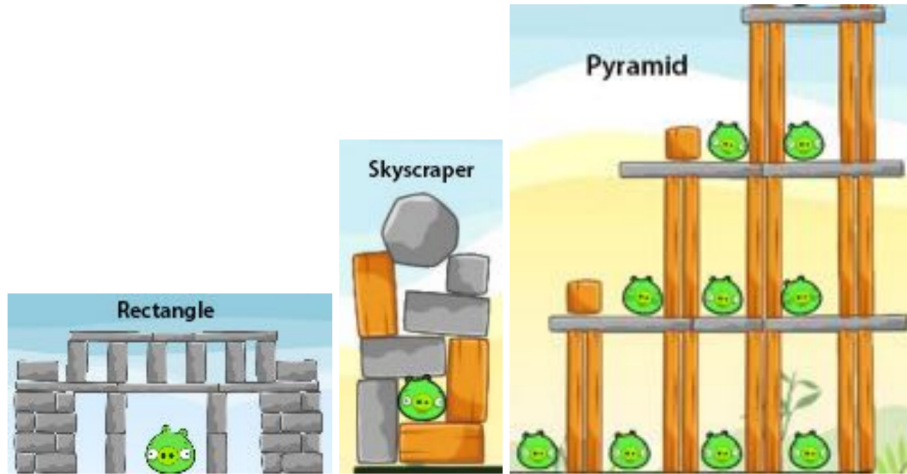


Figure 3.5: Different building illustrations

For pyramids it is usually better to shoot low in the hope to destroy a target block leading the the whole building collapsing. Skyscrapers require a more central shot, aiming to tilt the building entirely. Rectangles are a tricky combination of the two others.

Round object strategy

This strategy targets the round object directly or the block keeping it stable. This is done in the hope of releasing the "killer object" on the rest of the objects in the level until it has no energy left. In most of the cases just freeing the round object is enough, it usually has enough potential energy on it's own to clear the level. An example is shown in figure 3.6, breaking the ice will free the boulders that can clear the level.



Figure 3.6: Round object example

Destroy as many pigs a possible

This strategy does exactly what the name says, it is used as default when the other strategies don't seem to apply. It uses the trajectory with as much as possible pigs in the ways.

3.5.2 Competition results

The agent and managed to place second of the benchmarks (pp.23 and 24) for their first participation in 2014.

3.6 State of the art

As we can see, there are a lot of different approaches employed by the agents entering the competition. Some are simple and general, others get really specific. A selection was made to present the best agents of each "category" that performed decently in the benchmarks. There are probably more agents that would deserve to be presented but not all agents disclosed their strategies, and for some we were even unable to locate a description of their agent.

The best performing agent in the current benchmarks, for which we have information, is the agent that chooses between four specific strategies. Those strategies have been developed by trial and error by their team and hardcoded. Our goal is to create an agent that can actually learn and adapt on its own.

Chapter 4

First Agent

We started the development of this first agent, which had as purpose to get a decent score, this is after we already got a good feeling of the game mechanics, understanding what works, and identifying the reasons for it which helped us with some initial parameter tuning and decision making.

4.1 First Algorithm Idea

Since Angry Birds levels are mainly composed by blocks of different materials stacked on each other, it seems a good idea to start looking into dependency structures. We considered using dependency graphs, aiming to model what pieces of the world would be impacted if a given piece was hit by a given bird. But the Angry Birds world is not necessarily deterministic, hence the same shot might cause slightly different outcomes. Thus it seemed logical and fair to include probabilities of making objects fall over, breaking them, or pushing other items. Parameters that should have an impact on those probabilities include the type of bird in use, the materials constituting the different objects and structures, etc. After discussion, we also realized that including some machine learning to tune the probabilistic parameters would be a great addition.

At the beginning of the development, we started small and incrementally grew the agent. When we started to code our first agent, we designed it around a heuristic that computes a score for each block in the scene. Our heuristic then returned the block with the highest score in order to shoot it with the current bird.

4.2 Preliminary improvements

We made some improvements and some additions to the framework provided because the framework is incomplete and basic. First we modified the class *ABObject.java*, this class represents all the objects in the game any type of piece : {Pig, Bird, Stone, Ice, Wood, ...}. We added to this class three methods to help us to deal with the position of each block :

- **touches** : This method returns a boolean. It is true if the object touch the argument object otherwise false. In order to do so, each object was enlarged by 1 pixel. Then, the presence of intersection between them was verified.

```

/**
 * @return true if this object touches a
 */
public boolean touches(ABObject a){
    if(this.equals(a))
        return false;

    if(shiftArea == null){
        Rectangle mbr = (Rectangle) this.getBounds().clone();
        mbr.grow(shift, shift );
        this.shiftArea = new Area(mbr);
    }

    Area largeA = (Area) shiftArea.clone();

    if(a.getShiftArea() == null){
        Rectangle mbr = (Rectangle) a.getBounds().clone();
        mbr.grow(shift, shift );
        a.shiftArea = new Area(mbr);
    }

    largeA.intersect (a.getShiftArea());

    return ! largeA.isEmpty();
}

```

- **isSupportBy** : This method also returns a boolean. If the argument object supports the actual object it will be true. To do that we check that the two objects touch each other and that the supporter touches the actual object from below.

```

/**
 *
 * @param o
 * @return true if o supports this
 */
public boolean isSupportBy(ABObject o){
    if(this.equals(o))
        return false;

    int thisDown = y + height;
    int thisRight = x + width;

    int oRight = o.x + o.width;

    if(!touches(o) || thisDown - shift > o.y

```

```

        || (oRight - shift < x)
        || (o.x + shift > thisRight))
    return false;

    return true;
}

```

- **isOnGround** : this method checks if the actual object is on the ground or on a hill. To do that we check whether the object touches the hill or if the object's coordinates are near the elevation level of the ground.

```

/**
 *
 * @param ground
 * @param hills
 * @return true if this object is on the ground
 */
public boolean isOnGround(int ground, List<ABObject> hills){
    if(this.getY()+this.getHeight() >= ground -this.shift){
        return true;
    }

    for(ABObject h : hills){
        if(this.touches(h)){
            return true;
        }
    }
    return false;
}

```

We also added two classes to the framework to help us in our task. The first class that was added is **RectangleAlgebra.java** (Page 76 in the annex section). This class was created to determine how an object is positioned relative to another object. This class is used in the heuristic of our first agent to get the leftmost supporter of a given block. For example, if we want to know if the object x is on the left on the object y we use this method:

```

public static boolean left(ABObject x, ABObject y){
    if(x.x < y.x)
        return true;
    return false;
}

```

The second class that was added is **UCLUtils.java** (Page 80 in the annex section). This class contains only one method that returns all the supporters of an object. This class was used in the heuristic of our first agent to get all the supporters of our best piece.

4.2.1 Others possible improvements

The framework is basic, this means that the provided module can sometimes have unwanted behaviours. To possibly get better results, we could have improved these module. We did not do that because we wanted to focus on the design of our agent first and might or might not be worth it.

Vision module

The vision module provided with the Naive Agent can have trouble to detect the different blocks present in the level. Sometimes, it regroups some blocks into one block. This might seem harmless but is a problem because our agent use this information while selecting targets. The goal is to get a good score and if the input information is incorrect, we get an erroneous view of the scene, leading to non optimal moves.

This is especially true in our case, since we chose block stability as the core of our agent's heuristic. Because of this specific weakness, the calculation of the destruction percentage in the scene used later when we develop our second agent is wrong. The vision module sometimes does not even return the same number of piece in the level. We believe this is often due to the rotation of block caused by the shots.

4.3 Our heuristic

In this part, we will describe the heuristic we designed for the first agent. Our heuristic returns the block that has the biggest score. To compute the score of each block, we go through the list of all blocks existing in the scene. Each block get a score of +1 for each other blocks it touches and a bonus score of +100 for each pig it is in contact with. Finally, the agent selects the block 'A' that has the biggest score.

After this, our agent has to compute the destruction weight according to the type of the bird on the slingshot and the type of our best target block 'A'. The destruction weight is a weight that is computed based on prior knowledge of the game (see the basics starting on page 9) and some testing on our end. It represents the destruction ratio between the type of one given bird and the type of one given block. The table below displays an example of the chosen ratios for the combinations of the current level.

```
public static final int destroyUtility [][] = {
    // Ice, Wood, Stone
    {1000,1400, 500}, // RED_BIRD
    { 700,1000, 300}, // YELLOW_BIRD
    { 550, 200, 100}, // BLUE_BIRD
    {1100,1000, 700}, // BLACK_BIRD
    {   0,   0,   0} // WHITE_BIRD
};
```

In the code above, the lines from left to right represent the blocks of type ice, wood and stone and the rows represent from top to bottom the type of the birds in

the order they will be shot from the slingshot. So, a red bird is better to destroy a block of type wood than a blue bird ($1400 > 200$). The white bird has a weight of zero for each type of block, because it is not the bird that must touch the target, but the egg dropped by the bird while it is flying.

When the agent has his best target block 'A' with the corresponding destruction weight in a map, it adds to this map the leftmost block that supports our best block 'A' with its respective destruction weight.

The destruction weight for the supporters are computed in the same way but we add a bonus of 10, to give a slight edge to supporters. We consider the leftmost supporter because in this Angry Birds game the slingshot is always on the left of every level.

Because the sling is on the left and birds are always shot to the right, the left supporters are usually those we have to look at, they are more likely to be reachable, and can be used to bring the whole structure down. That is why the agent adds the leftmost supporter to the map and does so recursively until reaching the ground. Note that the ground cannot be altered and is thus always stable.

The best target is then selected among the blocks in the map based on the destruction a shot on that block would cause.

In the end, the agent has a map that contains many entries. Each entry is composed of a key identifying the block and the corresponding destruction weight. The key is a block representation using the Java class 'ABObject.java' that was provided in the framework. Our heuristic returns the block with the greatest destruction weight presents in the map, and our agent uses this block as target for the shot.

4.3.1 Heuristic Pseudo-code

```

for every block on the scene do
    // the bird currently on the sling is known
    Assign a basis score base on type of bird vs material
    for every other block do
        if they touch then
            | add 1 to the score
        end
        if this block is supported then
            | add 10 to the supporter's score
        end
    end
    for every pig do
        if they touch then
            | add 100 to the score
        end
    end
end

```

Algorithm: Heuristic pseudo-code

4.3.2 Example

In the output below and in the simple example in figure 4.1, we can see the object A returned by the heuristic. This is the object with the highest destruction weight, in this particular example it gets a score of 1410. We have a Red bird on the slingshot so the destruction weight is 1400 and this block is a support of the pig so it gets an addition of 10 to its destruction weight for a total of 1410.

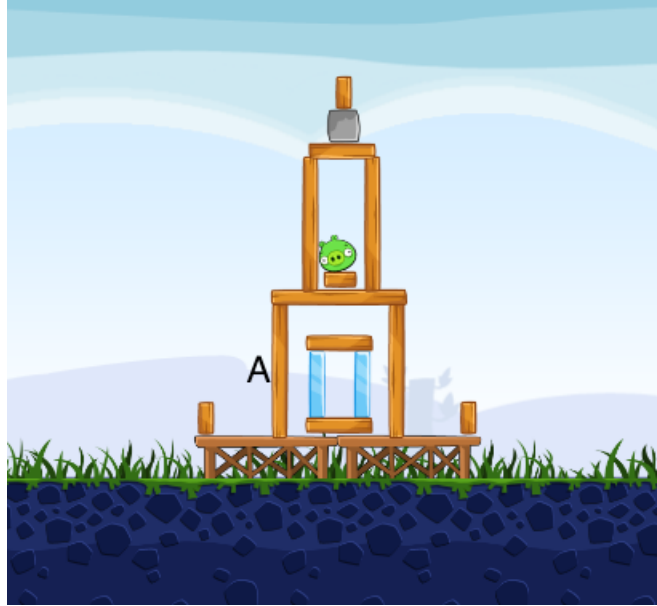


Figure 4.1: Example of the heuristic output

ABObject [id=136, type=Wood, area=0, shape=Rect, angle=0.0, hollow=false, x=519, y=315, width=2, height=50] Score = 1410

4.4 First agent pseudo-code

Our first agent is simple, it simply uses the heuristic on all available blocks in the level and shoots at the one obtaining the highest score.

```
while level not cleared or failed do
  for every block on the scene do
    Compute the heuristic value
    Remember the best block
  end
  Shoot at the best block
end
if level cleared then
  | Go to next level
else
  | Retry level
end
```

Algorithm: First heuristic based agent pseudo-code

4.5 First results

Our first agent with this simple heuristic manages to place 13th on the benchmarks of 2014 (Tables available pages 23 and 24 of the Basics section). It reliably obtains a score close to 892180. For comparison the Naive Agent provided with the framework by the competition website[1] gets a score of 756680 when it was executed it that year (note that the agent shoots randomly, it was not modified and scored 858730 the previous year). So, our agent scores slightly better but more importantly does not suffer from random "bad" luck. As a reminder, the benchmark score is obtained by summing the best score the agent obtained on the 21 levels (screenshots available in the annex p 73) during the allowed time. There can be no human interaction with the agent when he plays.

At the level 3 (see figure 4.2 below), our first agent even obtained a better score than the best agent in the benchmarks (pp. 23 and 24).

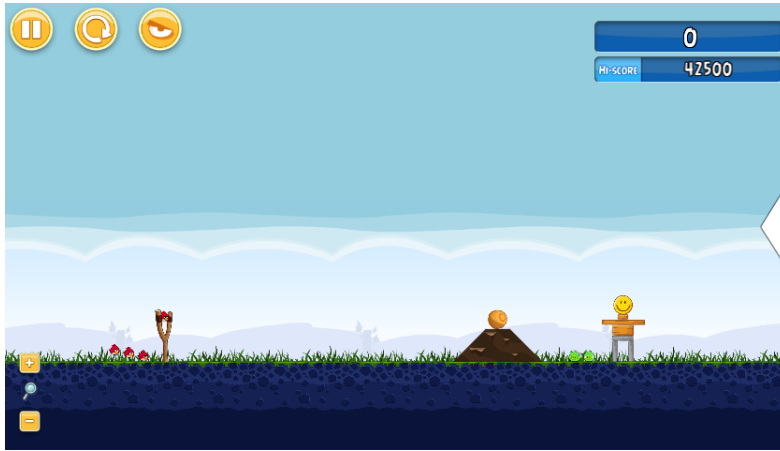


Figure 4.2: Poached Eggs : level 3

4.6 Limitations

Our first agent is limited, mainly because for the same level the heuristic will always returns the same best target. This could be a problem, it is indeed likely that there is another target that allows to win the level with less birds and hence a better score. Or worse, it could be that for some level the heuristic is wrong, and returns a bad target, the agent will always aim the bad target and might never clear the level. This did not happen in the test levels but we still need to consider the possibility of it happening.

The heuristic used in our first agent can be poor due to the fact it favors targeting the blocks that are in direct contact with the pigs and their supporters, but this is not always a good solution. Sometimes, there are no pieces that are in contact with pigs, and in other cases it can be beneficial to aim directly at the pigs instead of aiming blocks.

To make our first agent more clever, and to make it learn from its mistakes, we decided to improve it with a machine learning module. This is explored in the next section.

Chapter 5

Second Agent with machine learning

To improve our first agent, we expanded it with a module of machine learning. Before adding this module, we had to make some modifications to the heuristic of our first agent to make it random, which our first agent wasn't since it selects the best block as target.

We need randomness in order to collect data during the learning process of the agent, we need diverse shots. Indeed, if he always shoots at the same target, he would not learn anything of value. To include machine learning, we thus cannot use our first version of the agent in its current state. We need diversified data to allow the agent to learn to classify a shot as good or bad before shooting.

The goal is to have an agent that uses our first heuristic and that can use a classifier to predict if a candidate shot will significantly advance us in the task of clearing the current level. If the shot is good we take it, else we explore other candidates.

5.1 Modification of the first agent

To get a good set of data we redesigned our first agent so that it selects randomly from the blocks in a set containing the top scoring blocks with respect to our first heuristic, or all the pigs existing in the scene. This allows for possibility for direct shots targeted at pigs our first heuristic never considered.

There is a list of all the modifications we made in the second agent below:

- We randomized the selection of the targeted object. In the first agent, we selected the best piece of the scene as target. Now, in the second version virgule, the agent chooses randomly between all the best blocks and all the pigs in order to get one block or one pig for the shot. We have added the pigs in the set of selection, because with more experience on the game's mechanics, we have come to realize that sometimes it is better to aim directly the pigs, and not the structure as we can see on the figure 5.1. In this level, it is useless to push the structure to the right, remember the slingshot is always on the left of the scene, if the structure falls to the right no damage will be dealt at all to the pigs.

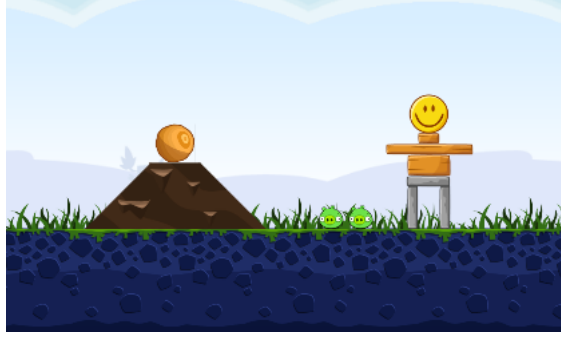


Figure 5.1: Level where it is better to aim directly the pigs.

- We also added a randomness to choose the target point between the centre of the piece or the upper left corner. In our experience, the upper left corner is the most effective in tilting structures. Again remember the slingshot position.

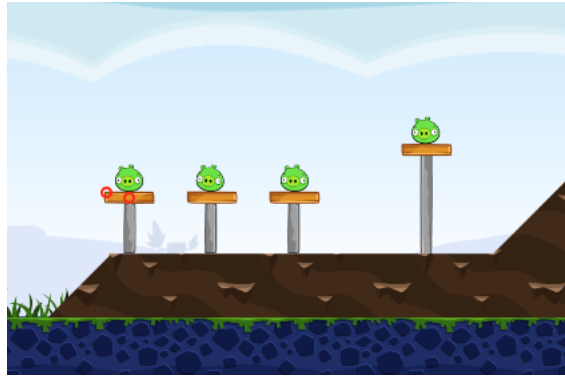


Figure 5.2: Random between center of the piece or the upper left corner

- We also randomized whether the agent chooses a high trajectory or a more direct shot. High trajectory shots can sometimes directly hit targets direct shots cannot.
- To get the tap interval when the agent should activate the power of the bird we also used a randomization according to the type of bird on the slingshot. Those are the interval we chose :
 1. Red bird: these bird have no power so the interval is 0%.
 2. Yellow bird : if we have a low ($< 45^\circ$) angle in the trajectory we have a random interval between 70% and 80% of the trajectory. If we have a high shot with a high angle in the trajectory we have a random interval between 80% and 90%.
 3. White bird: has the power of the white bird is to drop an egg, we need to activate it at the end of the trajectory. Thus we chose a random interval between 80% and 95%.
 4. Black bird: since the power of the black bird is to explode, we need to activate it when we are near the target. If we have a low ($< 40^\circ$) angle of the trajectory we chose a random interval between 90% and 100% but if we have a high shot we choose to activate the power at 100% of the trajectory.

5. Blue bird: the blue bird's power is to spits up to three birds, each dealing the same damage as the initial bird. We chose to activate it at a random interval between 65% and 80% of the trajectory.

5.2 Machine learning module

Now that these modifications included randomness into the new agent to support learning, we added the machine learning module. Uwing the **Weka** [7] library, we built a classifier to predict whether a shot will be **bad** or **good**.

5.2.1 Weka library

Weka [8] is a collection of machine learning algorithms for data mining tasks. The library has the advantage to allow us to call it directly from our own Java code. Weka also comes with tools for data pre-processing, classification, regression, clustering, association rules, and visualization.

The name comes from a flightless bird, of the same name, only found on the islands of New Zealand. Weka is an open source software issued under the GNU General Public License developed by the university of Waikato.



Figure 5.3: Weka

The Weka library uses an "arff" file format to store the learned data. This format has two sections : a **header** with the information on the file and the **data** section. The header section contains the name of the relation that will be saved in the file and the attributes of the data and their types. The data section contains the data saved in the file, one line by shot.

An example of our file with two shots can be found below :

```
@relation aibirds

@attribute numberShot numeric
@attribute pointX numeric
@attribute pointY numeric
@attribute typeBird {RedBird,YellowBird,BlueBird,BlackBird,WhiteBird}
@attribute typeTarget {Ice,Wood,Stone,TNT,Pig}
@attribute isReachable {true,false}
@attribute numberSupporters numeric
@attribute level numeric
@attribute tapInterval numeric
@attribute targetCenter {true,false}
@attribute rating {?,bad,good}

@data
0,612,244,RedBird,Wood,true,2,1,0,true,good
0,546,194,RedBird,Pig,true,0,2,0,true,good
```

We created a java object "UCLShot.java" to record the shot and save it in the "arff" file. It contains the same attributes as the example file above. We will explain with more details the different attributes in a following section.

5.3 Naive Bayes classifier

A Naive Bayes classifier is a group of simple probabilistic classifiers based on the Bayes theorem with independence assumptions between the features. The classifier is a model that **assigns a class label to a problem instance** where the class labels are drawn from some finite set.

The naive Bayes is a conditional probability model. The model assigns to a problem instance represented by a vector $V = (x_1, \dots, x_n)$ representing some n features probabilities for each of K possible classes C_k . To decide which class to assign to the problem instance, the model selects the class with the highest probability :

$$y = \underset{k \in \{1, \dots, K\}}{\operatorname{argmax}} \quad p(C_k) \prod_{i=1}^n p(x_i | C_k) \quad (5.1)$$

We chose the Naive Bayes classifier of the **Weka** [8] library because a Naive Bayes classifier assumes that the value of a particular feature is independent of the value of any other feature. This is necessary for the Angry Birds game, because for example, the type of target is independent of the coordinates x and y of the target.

The Naive Bayes classifier considers the features to contribute independently to the probability that the shot is good or not, regardless of any possible correlations between each features. An other advantage of the Naive Bayes classifier is that it requires a small amount of training data to estimate correctly the class of a new instance. Since learning is not easy for our problem, due to all the constraints on running the game etc, this was happily welcomed. Another reason to favor the Naive Bayes Classifier, is because experience tells us that this is one of the most practical learning methods.

5.3.1 Comparison with other classifiers

To compare multiple classifiers we decided to make a training set with half of the levels. The odd levels are training data and we test the classifiers on the even levels. We used this repartition because levels get significantly harder when the level number increases. For the learning task, we let each model play on the odd levels until they cleared it. For the comparison test task, we played every even levels once, thus if a model fails one level it gets 0 as score for this level.

Naive Bayes classifier

The scores obtained for the Naive Bayes classifier are given in Table 5.1. The scores obtained with the training set, corresponding to odd levels, are given in the left part of the table while the scores obtained for the test set, corresponding to even levels are given in the right part of the table.

Naive Bayes			
Training data		Test data	
1	30390	2	33350
3	41870	4	10890
5	41220	6	26430
7	21290	8	33750
9	34520	10	38330
11	56570	12	56040
13	38010	14	53280
15	38780	16	51410
17	49500	18	47050
19	41810	20	32980
21	67920		
Total training	461880	Total test	383510

Table 5.1: Naive Bayes classifier results for training and test data

Random Forest

Table 5.2 gives the score for the training set on the left, the odd levels and the score for the test set on the right, the even levels for a Random Forest classifier.

Random forests are a set of learning method for classification, regression and other tasks, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes or mean prediction of the individual trees.

Another constraint is that a Random Forest classifier needs some data before being able to predict a new instance which is problematic for us because, at the beginning of a new level, we might not have any available data.

Random Forest			
Training data		Test data	
1	30390	2	26450
3	42570	4	19240
5	46170	6	15940
7	28830	8	40040
9	49900	10	0
11	39570	12	54560
13	41620	14	57370
15	48230	16	55990
17	48630	18	49090
19	33620	20	47060
21	50960		
Total training	460490	Total test	365740

Table 5.2: Random Forest

Decision tree : J48

Table 5.3 gives the score for the training set on the left, the odd levels and the score for the test set on the right, the even levels for a decision tree classifier, C4.5 algorithm.

This is an algorithm used to generate a decision tree. The J48 is an open source Java implementation of the C4.5 algorithm in the library Weka. This model does not suit us because it gets a score of 0 for three of the unknown levels, meaning that it does not perform well in unseen cases.

J48			
Training data		Test data	
1	32770	2	43770
3	41870	4	11040
5	57350	6	35390
7	25440	8	55810
9	31620	10	0
11	42100	12	55570
13	29950	14	0
15	35900	16	60350
17	48730	18	0
19	30410	20	48230
21	58480		
Total training	434620	Total test	310160

Table 5.3: J48

Sequential minimal optimization (SMO)

Table 5.4 gives the score for the training set on the left, the odd levels and the score for the test set on the right, the even levels for a Sequential minimal optimization algorithm (SMO).

SMO is an algorithm for solving the quadratic programming problem that arises during the training of support vector machines. A SMO classifier needs some data before being able to predict a new instance. Again, this is problematic for us, because we might not be able to give data at the beginning of the game. This model performs the worst of all, getting a score of 0 for 5 of the unknown levels.

SMO			
Training data		Test data	
1	30390	2	25940
3	40500	4	28050
5	45300	6	0
7	25530	8	56590
9	48880	10	38770
11	49820	12	59040
13	34950	14	0
15	44080	16	0
17	46670	18	0
19	39770	20	0
21	69710		
Total training	475600	Total test	208390

Table 5.4: SMO

We narrowed it down to those classifiers because they are the ones being supported natively by Weka.

5.4 Attributes

We built our model using the following features, they allow Weka to build a model for our problem:

- The number of the shot in the level as numeric value. We chose to add the number of the shot because the position of each block and pig in the scene depends on the number of the shot and usually the position of each blocks and pigs is almost the same according to the level and the number of the shot.
- The coordinates of the target point, as numeric value for both X and Y.
- The type of the bird on the slingshot. It is represented by one of these : Red-Bird, YellowBird, BlueBird, BlackBird, WhiteBird. It is important because the classification of a shot depends of the type of the bird on the slingshot because some birds are better to destroy some type of block.
- The type of the targeted object is represented by one of these : Ice, Wood, Stone, TNT, Pig. It is important because the classification of a shot depends of the type of the target object as some birds are better to destroy some type of block.

- If the target object is reachable or if there are some objects on the trajectory. We added this feature to teach our model that an unreachable block or a pig is in most cases a bad shot.
- Numbers of objects that support the target object. We used this feature because when we aim a block that has zero or few blocks that supports it, it means that the block is at the bottom of the structure. A shot that aims a foundation target is more likely to bring down the whole structure and so destroy more blocks and kill pigs.
- The number of the level. Each level has his own structure of blocks and pigs. So to help the model on level that have already been played, we choose to add the number of the level. We later removed this feature, because it seemed unfair to give even more information for known levels.
- The distance of the total trajectory when we active the power of the bird in percentage value. We had to include this feature because of the importance of the bird's powers. Two identical shots with the same attributes but with a different activation time will often result in a good and a bad shot according to the type of the bird.
- If we aim the centre of the target object or the upper left corner. Sometimes, it is better to aim the centre of the block while sometimes, it is better to aim at the upper left corner so we added this choice to the model.
- The class attribute of the shot : bad, good or '?'. We used the '?' symbol when the model predicts if a shot will be bad or good. When we predict a new shot, we use a copy of the data as temporary data and we add to it the new shot with a '?' as class.

5.5 Shot classification

By default, all shots are classified as bad shots. A bad shot can become a good one when it fulfills one of these conditions :

1. The shot destroys more than 10% of blocks in the scene. We chose 10% because after some trial and error on test levels, it seemed a realistic percentage of blocks to destroy with one shot, still having an impact on the level. In the chapter 7, we will explain some ideas of improvements for our agent, one of whom is talking about this subject.
2. The shot is good when it kills a satisfactory number of pigs. A satisfactory number of pigs is the ratio between the number of pigs in the scene and the number of birds we have left (equals to the number of shots we have) and by default it is one.

```
double pigsNeeded = 1;
if (initPigSize > initBirdSize){
    pigsNeeded = Math.floor(initPigSize / initBirdSize);
    if (pigsNeeded == 0){
        pigsNeeded = 1;
    }
}
```

3. A shot is good when it wins the level.

When a shot is classified by the Naive Bayes classifier as a bad shot, we reject it and we restart the heuristic to get a new set of blocks and pigs to select a random new target. After each shot, the agent waits for a certain time, computes if the shot is bad or good and adds the results to the data. If this happens, it means our heuristic did not manage to provide good targets and thus that we have to allow more possible targets.

Birds are shot with a random trajectory selected between high and low. After every shot is made and waiting for a relatively long delay, the agent compute if it was a bad or good shot and adds the result to the data. The delay is there to let the game physics do their work, when a bird hits a structure it can take time for structures to fall and chain reactions to happen. If we do not wait long enough, the score can be worse than what we would normally get. We thus have to wait long enough for all action to end, but not too long to save playing time. In many cases, the random shot after three rejected targets, leads a bad shot but the agent can be lucky, leading to a good shot.

5.6 Destruction threshold

To classify if a shot is good or bad, we used three conditions. One of these is that the percentage of destruction of the shot must be greater or equal to a destruction threshold. After each shot, the agent computes his percentage of destruction with the following computation:

```
double destructionPercentage = (((double) (initBlockSize - vision.  
findBlocksRealShape().size())) / initBlockSize) * 100);
```

To compute the percentage of destruction of one shot, we divided the number of pieces present in the scene after the shot by the number of pieces present in the scene before the shot and we multiply this result by 100.

In some cases, the calculation is wrong because of the vision module which has problems with the detection of some blocks. It regroups multiple blocks into one same block, this is impacted by shots, when the blocks have moved the detection gets better or worse. This can leads to erroneous percentages because the module detects more block after the shot even if this is not possible a shot cannot create block in the scene but only destroy them.

5.6.1 Threshold : 10%

First, we tried with a threshold of 10% but we quickly noticed that the agent accepted too much shots. The agent accepts shots that are not so relevant: shots that do not earn enough score nor help to clear the level. As the threshold is relatively low this it was too easy to reach. This has however the positive effect of significantly reducing the chances of rejecting shots making the agent shoot randomly.

Despite this, the agent obtained excellent scores during testing, beating the best agent we had at that point, even without extensive learning. Compare to the benchmarks of 2014 (pp. 23 and 24), the two agents on the table 5.5 come in the

4th position even if the agent that played 5 times every level gets a slightly better score.

	10% : 2 playthroughs		10% : 5 playthroughs	
Level	Score	Tries	Score	Tries
1	29080	1	28730	1
2	42920	1	52950	1
3	40490	1	41870	1
4	20260	1	27910	1
5	57870	1	63300	1
6	35050	1	33210	1
7	28250	3	19470	2
8	48270	1	36530	4
9	42880	1	49310	1
10	52530	2	69170	3
11	46510	1	46480	1
12	58940	1	53600	1
13	44760	1	42880	2
14	68330	5	85570	1
15	42700	1	36110	4
16	58850	1	48870	1
17	49980	1	47750	2
18	47750	2	43450	2
19	40370	3	31590	3
20	40110	3	51590	1
21	68400	2	59440	2
Total	964300	34	969780	36

Table 5.5: Scores obtained by a learning agent for destruction threshold of 10%

The first column gives the current level (1 to 21 levels of the game p.73), the second column gives the score obtained for each level and the last one gives the number of attempts needed to pass the level. Table 5.5 is composed of two set of data. The first one is the obtained data for an agent that had two playthroughs on each level to learn, the second agent had five.

We tried thresholds lower than 10%, but they resulted in the agent shooting mostly randomly since any and all shots where classified as good, leading to no learning and bad performances. We thus set the lower limit a 10%.

5.6.2 Threshold : 20%

As we noticed that an agent with a 10% destruction threshold is a bit lax, we decided to increase the threshold to 20%. With this threshold, it becomes harder to get a good shot. As it is more difficult to find an acceptable shot, the agent falls more often in the phase of denial, and thus shoots mostly random shots. This is the opposite of thresholds below 5% but results in the same: a lot of shots are random.

In the worst cases, the agent shoots only random shots because it rejects all the proposed targets. This leads to a lot of wasted birds on random shots, which generally are bad shots. The consequence of this is to drastically increase the number of

attempts to pass the level which also increases the time needed to pass all the levels.

Agents with a threshold of 20% struggle to complete the 21 levels in the allocated time. Usually they do not have time to even attempt the last levels. We cannot realistically expect a single bird to break more than a fifth of the level's blocks in a single shot, thus we didn't explore thresholds above 20%.

	20% : 2 playthroughs		20% : 5 playthroughs	
Level	Score	Tries	Score	Tries
1	32770	1	29080	1
2	34270	1	52290	1
3	41240	1	41870	1
4	29620	1	19720	1
5	56390	2	56400	1
6	15260	1	34720	1
7	31960	3	30720	8
8	25100	1	26880	2
9	32220	1	38390	1
10	56980	1	64390	1
11	48980	1	51450	1
12	59550	1	52700	1
13	41720	1	46550	1
14	54090	2	64420	2
15	47930	3	41030	2
16	58310	1	48720	1
17	50100	1	43010	4
18	48470	1	40770	5
19	29000	13	34940	1
20	60020	8	42490	2
21	63410	4	62380	11
Total	917390	49	922920	49

Table 5.6: Scores obtained by a learning agent for destruction threshold of 20%

Table 5.6 is composed of two set of data. Again, the two data sets represent a learning phase of two and five playthroughs on each level. The first column gives the level number (1 to 21 levels of the game p.73), the second column gives the score for each level and the last one gives the number of attempts to pass the level.

Compared to table 5.5 it can be seen that with a higher destruction threshold more tries are made and more time is needed to complete all the levels. However, the score does not increase. In conclusion, a higher threshold does not give better results.

5.6.3 Threshold : 15%

Given the results of the previous destruction thresholds, we decided to test a threshold between 10% and 20%. We thus chose an intermediate threshold of 15%. This is a good compromise for the destruction percentage as the agent does not accept too many irrelevant shots nor refuses too many shots.

	15% : 2 playthroughs		15% : 5 playthroughs	
Level	Score	Tries	Score	Tries
1	30390	1	29080	1
2	43590	1	52450	1
3	42280	1	42240	1
4	28220	1	28390	1
5	65560	3	60490	2
6	17060	2	35570	1
7	28310	1	23510	3
8	57140	2	29550	1
9	49560	1	25070	1
10	44910	1	34550	1
11	31770	1	31120	1
12	57090	1	39630	1
13	32440	2	24900	1
14	52670	1	51390	4
15	49850	1	33810	1
16	46770	1	63230	1
17	45640	1	39830	1
18	41260	1	42140	4
19	31730	1	31930	1
20	42330	4	42100	4
21	59460	1	65420	3
Total	898030	29	826400	35

Table 5.7: Destruction threshold of 15%

Table 5.7 is composed of two sets of data. The first one represents the score and the number of tries for each level of the agent with a destruction threshold of 15% which during his training period played two or five times each level. The first column gives the level number (1 to 21 levels of the game p.73), the second column gives the score for each level and the last one gives the number of attempts to clear level.

By comparing Table 5.7 with the table 5.6, it can be seen that the agent wins the level with less tries which means that the agent wastes less time in each level. We can also see from table 5.7 that the agent obtained a smaller score when it played five times each level than when it played two times each level. This can partly be due to the randomness when selecting a target among all the targets returns in the set by our heuristic, but it seems significantly worse than a threshold of 10% we thus decided to keep.

5.7 Overfitting

When we designed the first version of our agent with the module of machine learning, we decided to include the number of level as one of the attributes. In fact, it was not a good idea because this attribute induced overfitting of the data as the agent could learn by heart the good shots for each level.

We thus decided to evaluate the effect of this attribute by comparing two versions of the agent: one with the attribute "level" and one without it. Each version performed 20 times each level, results are in the figure 5.4 below.

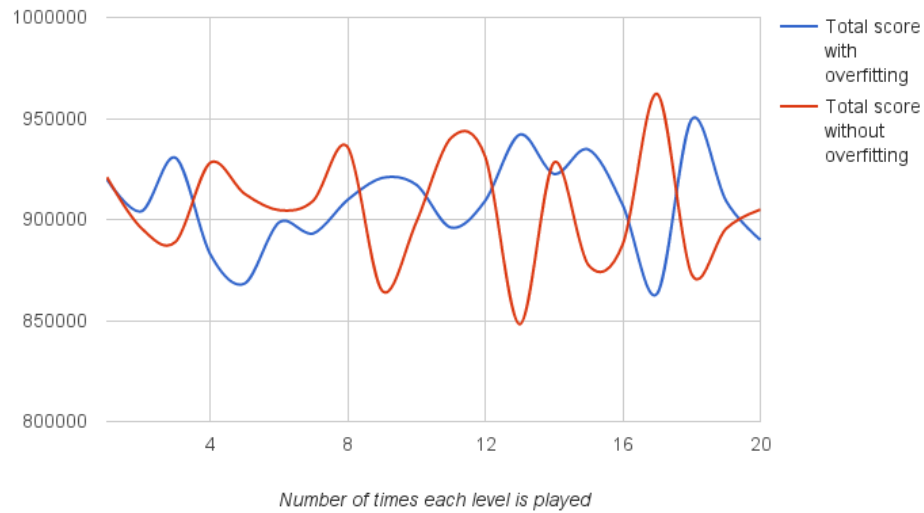


Figure 5.4: Level attribute with and without level attribute

On figure 5.4 the horizontal axis represents the number of times each level is played while the vertical axis represents the total score obtained by the agent.

The blue line represents the agent with the attribute `level`, this is the agent that overfits the data for each level. The red line represents the agent without this attribute. There is not really an agent that stands out compared to the other, the two curves are almost the same lines. Thus the level attribute was removed from our agent without noticeable performance loss.

5.8 Agent pseudo-code

In this section, we describe the functioning of the agent. There are two distinct part, the part used by the agent to obtain valuable learning data. This part forces the agent to make bad shots from time to time, to also have data for those. The second part is used to perform a good score, using all the data collected during the learning phase. It is worth noting that the agent continues to store the data from the shots made when playing.

5.8.1 Learning part of the agent

The main idea of our agent is that it is in an eternal learning process. That means that the more it plays, the more it learns, and hopefully the better it gets. Each time the agent plays a bird, it analyses its last shot and adds it to its data. The main steps of our second agent's learning part are the followings :

1. Select a target with the help of the heuristic of our first agent. As explained in the previous sections, either a block or a pig can be selected. Each target has the same probability to be selected. Once we have that target, there is another random selection, this time to decide whether to make a high shot or a direct one. Here the probabilities for a direct shot are set at $5/6$. The last parameter is the time before activating the bird's power, this is randomized based on the type of bird and the trajectory as previously explained.
2. Now that we have the randomly created shot, we predict the class of the shot.
 - If the prediction is good then the agent makes the shot. As a reminder, a shot is qualified as good if it destroys more than 10% of the blocks in the level, clears the level, or kills a significant fraction of pigs.
 - If the prediction is bad and it has been less than the third try select a new shot.
 - Else if the prediction has been bad three times in a row, it selects a random block or pig as target and shoots, again every block or pig has the same probability to be selected.
3. Analyze the last shot results with the shot classification.
4. Save into the data the shot and its class and remake the classification model.

These steps are repeated, for every bird in each level that the agent plays until the level is failed or cleared.

```

while the level has not been cleared or failed do
  for every available bird on sling do
    Select a target based on our heuristic
    Randomize trajectory
    Randomize tap interval
    Classify the shot
    if shot classified as good then
      | Make the shot
    end
    if shot is classified as bad and it has been 3 times then
      | Select a random target
      | Randomize trajectory
      | Randomize tap interval
      | Classify the shot
      | Make the shot regardless of classification
    end
    if shot is classified as bad and it has been less than 3 times then
      | // skip
    else
      | Add the shot to the data
      | Update classification model
    end
  end
end
if level cleared then
  | Go to next level
else
  | Try level again
end

```

Algorithm: Learning part of the agent pseudocode

5.8.2 Scoring part of the agent

This serious part is designed to achieve a good score, using the collected data to do so.

The pseudocode is very similar to the learning agent except that here, the agent can only select a target in the shots qualified as good and the pigs. This means that we cannot get a bad shot, we can thus immediately shot at the target after randomizing the shot's components and save the results, same as the learning part.

```
while the level has not been cleared or failed do
  for every available bird on sling do
    Use our heuristic to get the list of best target blocks
    /* the order of the blocks in the list is randomized */
    for every block of the list do
      Randomize trajectory
      Randomize tap interval
      Classify the shot
      if shot classified as good then
        Keep as target
        break
      end
    end
    if no target found then
      Select a random target
      Randomize trajectory
      Randomize tap interval
      Classify the shot
      Make the shot regardless of classification
    else
      Make the shot on target
    end
    Add the shot to the data
    Update classification model
  end
end
if level cleared then
  | Go to next level
else
  | Try level again
end
```

Algorithm: Serious part of the agent pseudocode

5.9 Training set

We used as training set the first 21 levels of Poached Eggs section of the game. Screenshots giving a general idea of the 21 levels are displayed in the Annex section on page 73. Our agent played many times each level to improve its learning data, with various situations to not only have a single set of shots for each level.

5.10 Learning curve

The learning curve has been computed on the learning part of the agent with machine learning module. We decided to use the number of tries needed to complete the 21 levels as experience. For the time, we chose the number of playthroughs, meaning the number of times the agent managed to clear the 21 levels, one after another. The agent started with an empty data, and built it up while playing.

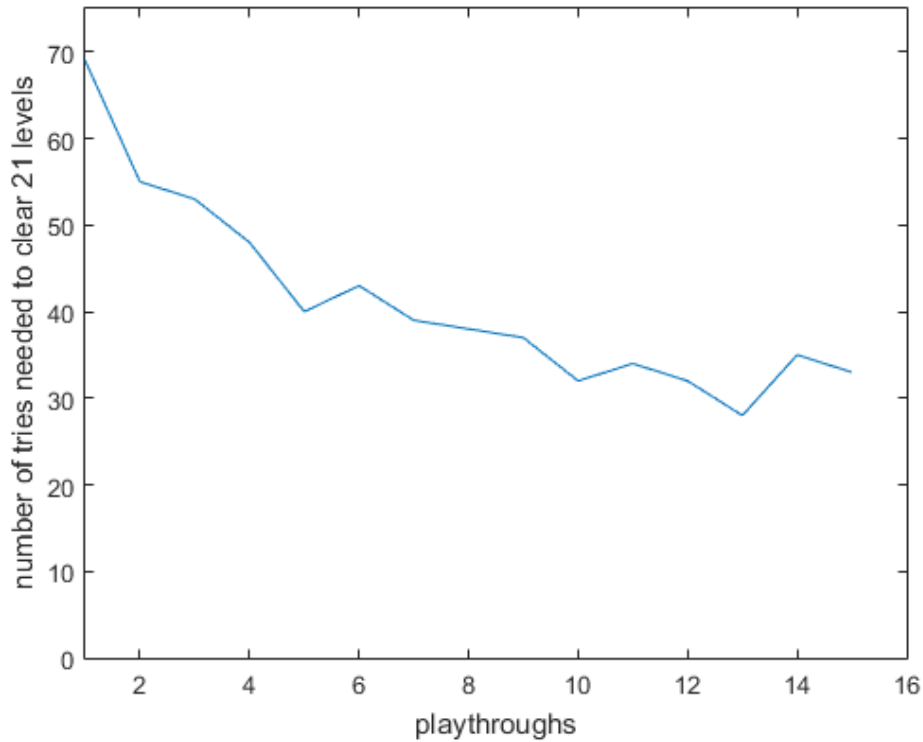


Figure 5.5: Learning curve

As we can see the learning of our agent is quite visible, the maximum number of tries the agent needed was 69 and the minimum was 28. We made a total of 15 playthroughs and monitored the evolution of the agent. We noticed that passed 10 playthroughs the agent's learning speed decreases and stabilises around 30 tries, the minimal number of tries needed to clear the 21 levels is obviously 21, if every single level is managed with a single shot.

5.11 Classification repartition for a set of shots

We tested the classification rate for the machine learning agent, the results we obtained are in table 5.8 below.

		real		
		good	bad	
prediction	good	132	43	175
	bad	83	33	116
		215	76	

Table 5.8: Classification repartition for 2 playthroughs

This might seem pretty bad at first, especially the Type II error is pretty high, meaning that there is a significant portion of the data that is classified as bad, that in reality is good. But our agent when playing seriously, only uses shots that are predicted as good. More problematic for us are the Type I errors, good shots that are predicted as bad. Those will have an impact on our agent, because they might actually be used.

The actual numbers that are important here are thus 132 shots correctly classified as good on a total of 175 shots. This is a bit above 75% of the shots that are correctly classified. It is also important to remember that a level is not constituted by only one shot, the player usually has more than four birds to clear the level. Of course we would ideally not want to waste any bird given the huge amount of bonus points they are worth, but errors are unavoidable.

Chapter 6

Comparing agents

Every time we finished developing an improved version of the agent, we made it learn by playing the game during multiple days. After this period, our agent had played many times each level and had computed numerous bad and good shots, allowing us to access its performances.

We then compare the score of six agents using table 6.1 page 65 the score of six agents : the basic Naive Agent provided with the framework by the competition website[1], our first heuristic agent, our final agent with the machine learning module. For comparison we added the first three agents of the benchmarks of 2014 (pp.23 and 24). We compared the scores obtained for the first 21 levels of the game.

In the benchmarks, it can be seen that the contest organizers added the score of the Naive Agent (version 2013 and version 2014) to obtain more data. We have executed the Naive Agent once, and obtained a total score of 869430 earning it a ranking of 16th in the benchmarks. Our first agent gets a total score of 892180 positioning it 13th in the benchmarks ranking . Our final agent gets a total score of 964020 ending in 4th position of the benchmarks. Finally the the first three agents of the benchmarks get respectively a total score of 1002380, 981120, 974670.

First, let's compare the Naive Agent with our first agent. Then we will compare our first agent and our final agent with the machine learning module. Finally, we will compare our final agent with the machine learning module with the first three agents of the benchmarks.

6.1 Naive Agent VS our first agent

First of all, we must say that the Naive Agent provided by the competition website is a random agent. This means that, when it has to select a target, it chooses a pig randomly, meaning each execution get a different score. We had to execute the Naive Agent ourselves to get additional data for comparison. The available benchmarks did not provide the number of tries that the agent needed to clear each level.

It can be seen in table 6.1 below, that the Naive Agent needs more or less the same number of tries before completing a level than our first agent. During preliminary tests, we had noticed that the Naive Agent usually does significantly more tries when it plays the game. But in the execution when we monitored for the comparison, we can say that it was lucky, and will probably not score better with only a few play troughs.

Our first agent is not random so he gets almost all the time the same score. Its score is better than the score of the Naive Agent as our first agent actually has a strategy, and aims the block with the biggest weight, in other words, the block that is in contact with the most others blocks and pigs in the scene.

6.2 Our first agent VS our final agent

Because our final agent with the machine learning module rejects shots he knows will be bad, the number of tries to complete a level decreases. The final agent does not play a bad shot for the three first targets that are provided by the heuristic, increasing his chances of clearing the levels early. Finishing early means also not using up all the birds which provide a significant point boost.

This is very important as during the contest each agent gets a limited time to play all the levels so if we do less try by levels it means that we don't waste our time, and can actually utilize this time to redo levels where the agent didn't perform well compared to other agents or hope improve a level where he did well.

6.3 Our final agent VS the first three agents

We can see on table 6.1 that on three levels (level 8, level 10, level 13) our final agent obtains a better score than the first three agents. We think that our final agent does a better score thanks to the machine learning applied to the base heuristic favoring strategical structure destruction.

We can see that on those levels the pigs are hidden behind hills or hard to break stone structures since it is very hard to break the stone, and not even possible to dent the hills, the idea for the level 8 is to get a shot around the hills and the stones to win the level. On level 10 and 13 the good shot idea is to manage to hit the easy to break ice blocks, some luck can help since the structure collapsing doesn't always destroy the pigs, sometimes the stones form an even harder to break bunker protecting them. Ideally we want to achieve this using the fewest possible birds to get a best score. During the learning period, our final agent discovered where to shoot and managed to succeed these levels with a higher score than the other agents because it directly makes a very good shot.

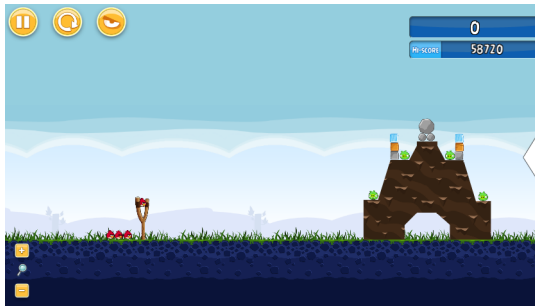


Figure 6.1: Poached Eggs : level 8

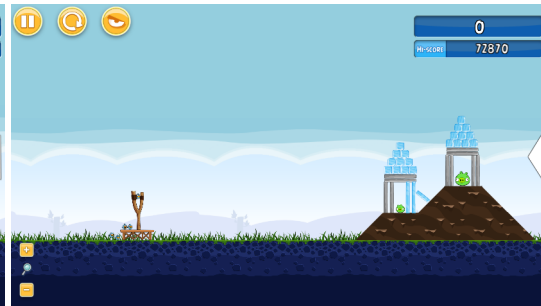


Figure 6.2: Poached Eggs : level 10

Level	NA score	UCL score	UCL-ML score	NA tries	UCL tries	UCL-ML tries	PlanA+: 1	DataLab: 2	AngryHex: 3
1	29160	30920	30760	1	1	1	30480	31620	30390
2	34160	34200	61410	1	1	1	62370	52000	54160
3	33510	41980	41900	1	1	1	40620	41890	41890
4	20710	27930	18770	1	1	1	29000	19790	28000
5	60070	56100	49580	1	1	1	69440	70320	64440
6	24850	25620	25530	1	2	1	36970	15700	24990
7	22500	22800	30050	1	1	1	32020	45720	36900
8	25750	38140	57890	1	2	3	47320	43190	24860
9	41570	36380	44900	3	1	1	26440	50420	49570
10	37040	58030	72870	5	2	1	56830	56790	50570
11	49680	40760	47760	1	4	2	47240	50650	53510
12	48650	47450	38970	1	1	1	58210	53420	57750
13	41540	34190	47580	2	14	1	34010	32010	42010
14	75430	66650	57010	1	4	2	65640	55640	58190
15	47610	35620	49030	1	4	1	54910	46450	46550
16	55990	62970	58800	1	1	1	57530	57380	63430
17	41490	43850	47870	1	1	1	51190	48570	46820
18	45380	40710	44460	1	1	1	52120	45730	50020
19	30240	34040	38480	2	2	1	39440	35470	38460
20	41900	41640	36100	7	6	3	45980	54680	46970
21	62200	72200	64300	7	1	2	64620	73680	65190
Total	869430	892180	964020	41	52	28	1002380	981120	974670

Table 6.1: Comparison table

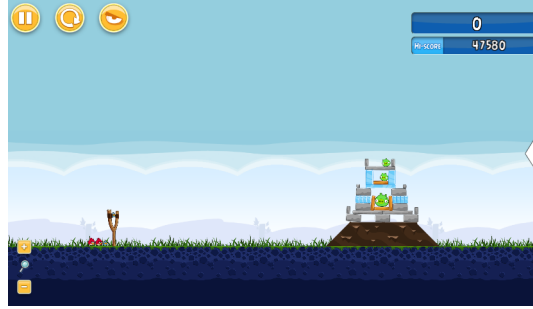


Figure 6.3: Poached Eggs : level 13

At the end, our final agent is just behind the first three agents at the 4th rank on the benchmarks. It means that, with more learning and maybe with a improvement of the heuristic, we could be in the top three. In the chapter 7, we will explain some ideas and leads on possible improvements for our agent.

Chapter 7

Possible improvements

As in every IT project, improvements could have been made, if we had more time. We compiled a list of some ideas we thought off that might improve our final agent. Updating the agent taught us that making the agent's heuristics very specific is not necessary the way to go. It lead to good shots in very specific cases, but for the other levels, that where more numerous tended to score much worse. The key is to find a sweet spot between generality and specificity. This means that, before testing it is hard to tell if the following ideas will have an overall beneficial impact or not.

- The features that we currently used for our classifier was the best combination we found during testing, but we think that the agent could be improved by new features or removing some by doing statistical tests. The problem is that it takes a long time for the agent to learn and we cannot easily swap a feature out to see what happens. The fact that we have to install plugins for chrome, prevent the website from reloading the cache for the now non existing game website complicates this task, making it hard to run on many machines simultaneously. For example, we could add the angle of the trajectory as a feature to make the agent learn about high trajectory versus direct shots.
- Improve the heuristic of the agent to get a better set of pigs and blocks to aim. We could, for instance, experiment with adding more weight for blocks that are in the vicinity of pigs because these blocks are more likely to touch the pig when they are hit by a bird, with a block on the left of a pig having a greater chance to fall on said pig etc. We tried this in a very basic form and it did not improve anything, but with more tuning we believe it could improve our agent.
- Find a better destruction threshold to qualify the shots than 10%. It is hard to find a percentage of destroyed blocks for every level that means the shot actually helped. The main reason for this is that even if blocks are destroyed, shooting at a structure for the sake of destroying it often results in the remaining blocks protecting the remaining pigs more than before. We tried multiples values but there might be a better value out there.
- After the first three rejected shots, when we chose a random block as target, maybe we could try to still classify the random shot and reject or find a better way to deal with it. If we know it will be a bad shot, we could avoid to waste a birds on a random bad block. In fact the random shot after 3 bad classifications was added in order to not exceed the time limit to clear the 21 benchmark levels.

- Make the agent learn with a specific agent for each level to get more good shot in our data. The specific agent for each level could be designed to only win one level and so get only good shot. The downside is that this would not help with new levels.
- Add a tactical module of some sort to replay level with the lowest score as our agent still had time when he reached the 4th place, if he utilised that time to try and redo a few levels he might have won a place in the ladder. This module should have access to the score of the other agents and thus allow to find the ideal levels to replay with the given time to maximize the score improvement.

Those improvements only focus on improving our existing agent. It should be possible to try mixing it up with other methodologies, in the lines of those proposed by other agents in the state of the art section. We just have to make sure not to waste too much time computing before making the shot, because the time needed for the game to simulate the game physics and waiting for all objects to stop moving after a shot takes up the bulk of the available time.

Chapter 8

Conclusion

To conclude this paper, we will make a short summary of what we have covered.

We started by looking over the different game components of the Angry Bird game we considered, the chrome version of the game. This version is also used as base for a AI competition that is held each year. The web based game is unavailable nowadays, slightly complicating the task. The unique constructions of the game, composed with the materials and bird's unique abilities make this a fun successful and challenging game.

But we were tried to build a good performing agent, so we had to take into account the competition framework to be able to compare comparable things. The agent we developed respects all the competition requirements, interacts through the chrome module with the game, allowing us to situate our performance based on the benchmarks the organisers provided on the competition website.

Articles on various subjects treating of the angry bird game helped us discover what had been done and tested. Some universities that participated in the competition wrote very detailed documentation for their agents, this helped us in the task of making a state of the art for the AI agents with the task of playing Angry Birds as effectively as possible. We discovered that there were many different approaches to achieve this goal, and we selected one that was never used, that we know of, or in any case never performed well in the competition.

We then proceeded to start the development of our agent. Experimenting a bit with the provided naive agent, altering it slightly and observing the impacts of various changes was a start. This allowed us to improve certain functions of the framework that did not work quite well. Our first agent was simple and heuristic based, it aimed at the theoretically optimal target block. This target was computed given our assumptions, the next bird that was to be shot from the sling, and the different materials that composed the structure protecting the pigs is the current level.

This agent was clearly an improvement when compared to the Naive Agent, that shoots randomly. But this agent did not perform exceptionally, notably struggling on level with huge amounts of little blocks.

Machine learning is what we turned to in the hope of making a smarter agent. With the help of the Weka package, we modified the first agent. It had to include

randomness, allowing to classify different shots, and evaluate various shot's performances. Different features were tested, this task was very time consuming since the agent had to learn for quite some time before performances really improved. Even the benchmark evaluation takes time, each level has to be completed and the physics of the game make it so it takes time before all the blocks come to a halt.

We proceeded to tuning various parameters of our agent, tried different combinations of attributes, changed classification thresholds, and evaluated the learning process.

The following section compares the benchmark performances of our agents and the top three agents in the currently available benchmarks. The learning curve confirms the agent actually does learn, and thus performs better with date. Different parameter tunings are studied to optimize performances. Our machine learning agent performs well, it never scores horribly compared to the others and even beats them from time to time.

Some improvement ideas are explored and conclude this paper.

Chapter 9

Bibliography

- [1] Angry birds ai competition. <http://aibirds.org/>.
- [2] Angry birds wiki. <http://angrybirds.wikia.com/wiki/>.
- [3] Stefan Schiffer and Maxim Jourenko and Gerhard Lakemeyer. AKBABA: The KBSG 2013 Team for the Angry Birds AI Competition. 2013.
- [4] Nikolaos Tziortziotis and Georgios Papagiannis and Konstantinos Blekas. A Bayesian Ensemble Regression Framework on the Angry Birds Game. 2014.
- [5] Joo-Seon Lee and Hyun-Su Seon and Jeong-Hyeon Kim and Su-Young Joo and Kyung-Joong Kim. Angry Plan A+ Team – Team Description Paper. 2014.
- [6] Tomas Borovicka and Radim Špetlík and Karel Ryměš. AI Competition: Data-lab. 2014.
- [7] Weka. <http://weka.wikispaces.com>.
- [8] Mark Hall and Eibe Frank and Geoffrey Holmes and Bernhard Pfahringer and Peter Reutemann and Ian H. Witten. The WEKA Data Mining Software: An Update. 2009.
- [9] Raúl Lara-Cabrera and Mariela Nogueira-Collazo and Carlos Cotta and Antonio J.Fernández-Leiva. Game Artificial Intelligence: Challenges for the Scientific Community. 2015.
- [10] Francesco Calimeri, Michael Fink and Stefano Germano, Giovambattista Ianni and Christoph Redl and Anton Wimmer. AngryHEX: an Artificial Player for Angry Birds Based on Declarative Knowledge Bases. 2013.
- [11] Du-Mim Yoon and Kyung-Joong Kim. Challenges and Opportunities in Game Artificial Intelligence Education Using Angry Birds. 2015.
- [12] Anjali Narayan-Chen and Liqi Xu and Jude Shavlik. An Empirical Evaluation of Machine Learning Approaches for Angry Birds. 2013.
- [13] Peng Zhang and Jochen Renz. Qualitative Spatial Representation and Reasoning in Angry Birds: The Extended Rectangle Algebra. 2014.
- [14] Angry birds ai competition. <http://chrome.angrybirds.com/>.
- [15] Example weka aibirds. <https://github.com/zackehh/weka-birds>.

- [16] S. Lin and Q. Zhang and H. Zhang. Object representation in angry birds game. *International Joint Conference on Artificial Intelligence*, 2013.
- [17] L. A. Ferreira and G. A. W. Lopes and P. E. Santos. Combining qualitative spatial reasoning utility function and decision making under uncertainty on the angry birds domain. *International Joint Conference on Artificial Intelligence*, 2013.
- [18] XiaoYu. Ge and S. Gould and J. Renz and S. Abeyasinghe and P. Zhang and J. Keys and A. Wang. Angry birds basic game playing software, version 1.31. *Technical report, Research School of Computer Science, The Australian National University*, 2014.
- [19] A. Narayan-Chen and L. Xu and J. Shavlik. An empirical evaluation of machine learning approaches for angry birds. *International Joint Conference on Artificial Intelligence, (2014)*, 2013.
- [20] Yves Deville. LINGI2261 Artificial Intelligence. 2016.
- [21] Pierre Dupond. LINGI2262 Machine Learning Classification and Evaluation. 2016.
- [22] Weka documentation manual. <http://www.cs.waikato.ac.nz/ml/weka/documentation.html>.
- [23] Stuart Russell and Peter Norvig. *Artificial Intelligence A Modern Approach (Third Edition)*. Pearson New International Edition.
- [24] Irina Rish. An empirical study of the naive Bayes classifier. 2001.
- [25] Random forest. https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm.
- [26] Platt John. Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines. 1998.
- [27] Quinlan J. R. C4.5: Programs for Machine Learning. 1993.
- [28] Quinlan J. R. Induction of Decision Trees. 1986.

Chapter 10

Annex

Poached Eggs : 21 levels

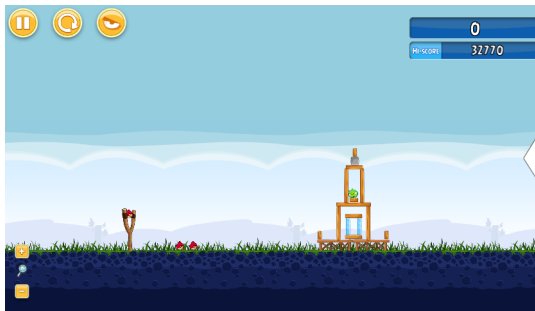


Figure 10.1: Poached Eggs : level 1

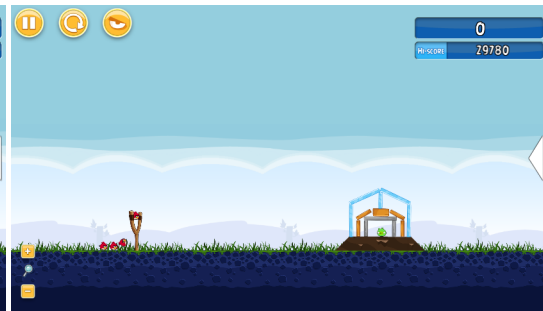


Figure 10.4: Poached Eggs : level 4

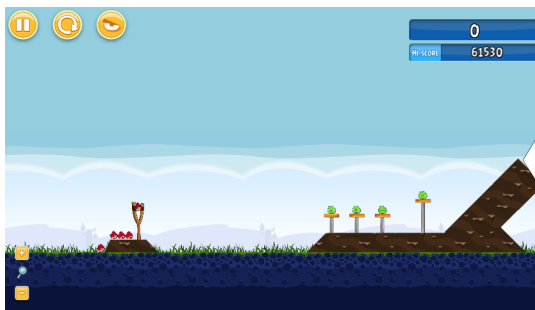


Figure 10.2: Poached Eggs : level 2

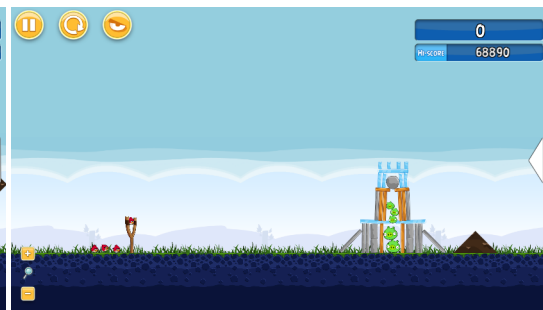


Figure 10.5: Poached Eggs : level 5

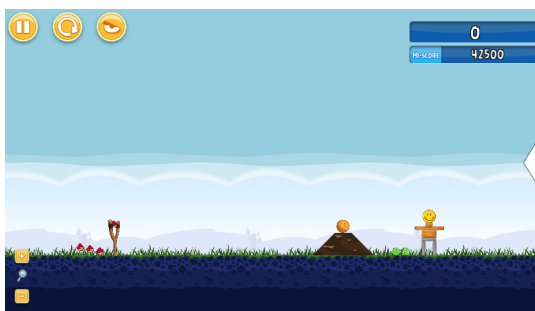


Figure 10.3: Poached Eggs : level 3

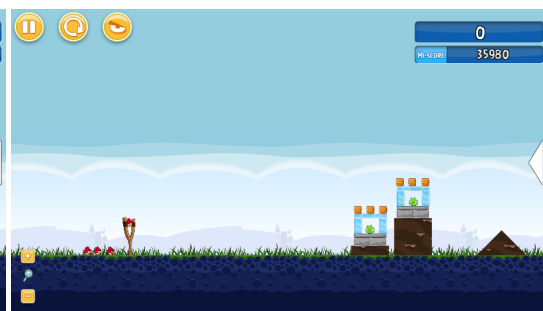


Figure 10.6: Poached Eggs : level 6

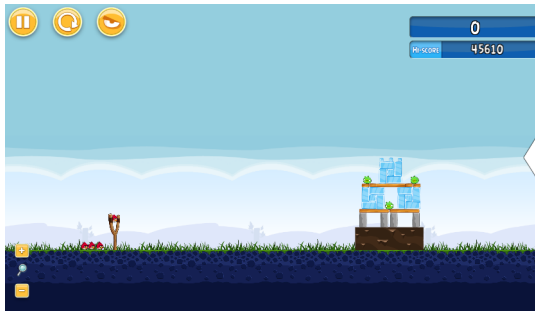


Figure 10.7: Poached Eggs : level 7

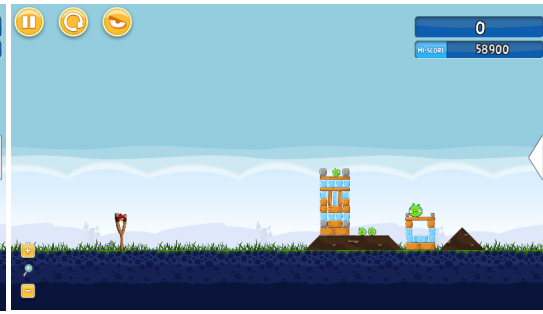


Figure 10.11: Poached Eggs : level 11

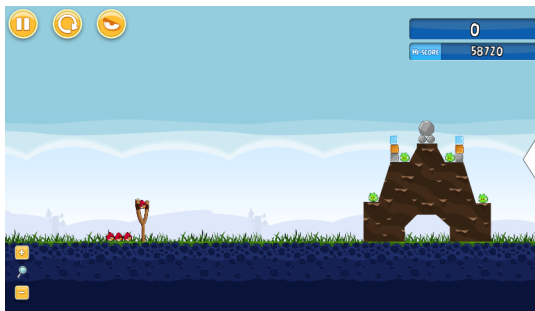


Figure 10.8: Poached Eggs : level 8

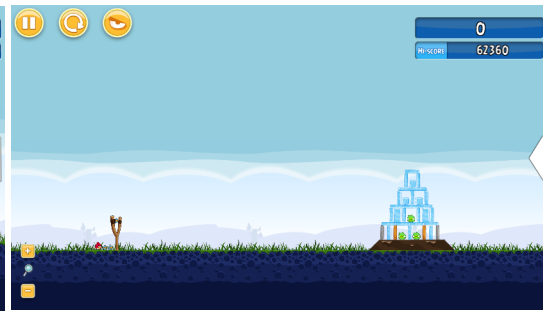


Figure 10.12: Poached Eggs : level 12

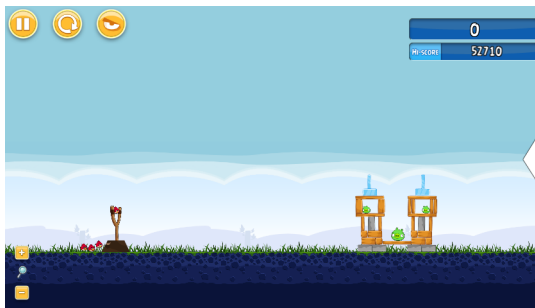


Figure 10.9: Poached Eggs : level 9

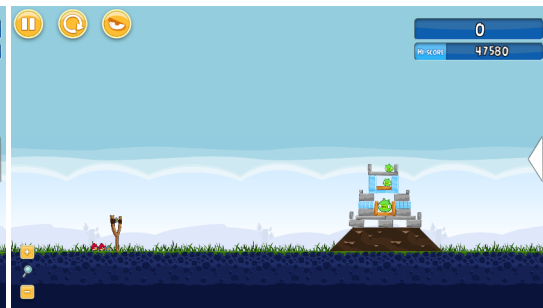


Figure 10.13: Poached Eggs : level 13

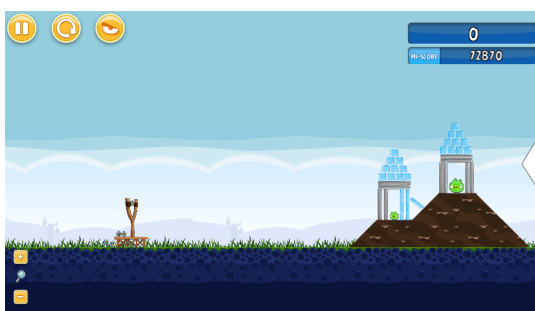


Figure 10.10: Poached Eggs : level 10

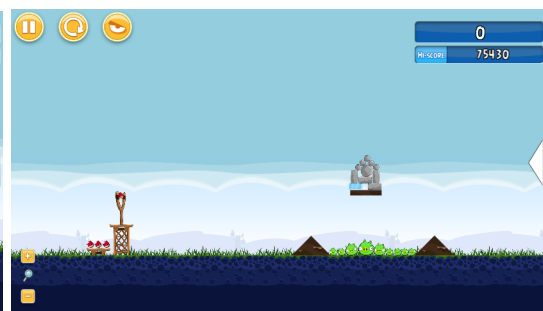


Figure 10.14: Poached Eggs : level 14

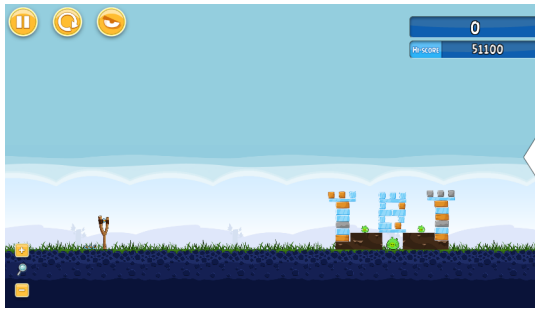


Figure 10.15: Poached Eggs : level 15

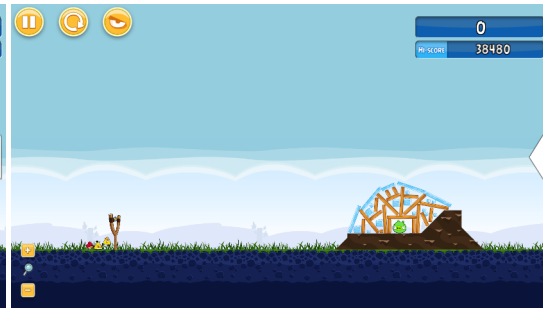


Figure 10.19: Poached Eggs : level 19

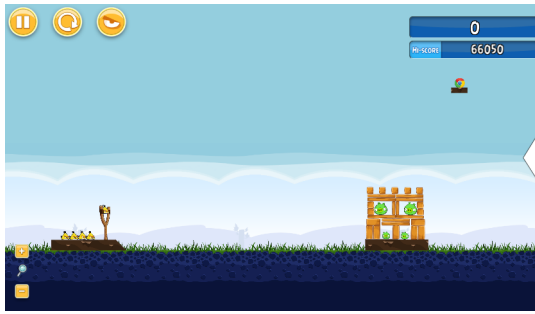


Figure 10.16: Poached Eggs : level 16

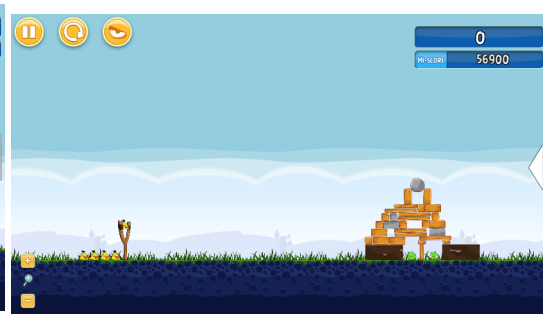


Figure 10.20: Poached Eggs : level 20

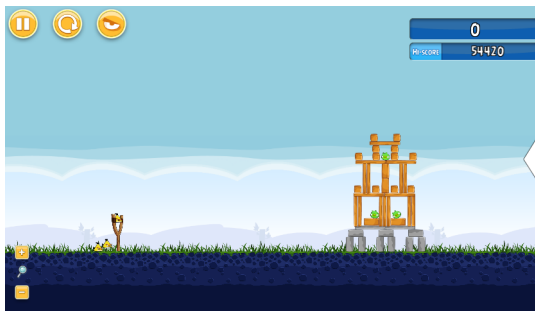


Figure 10.17: Poached Eggs : level 17

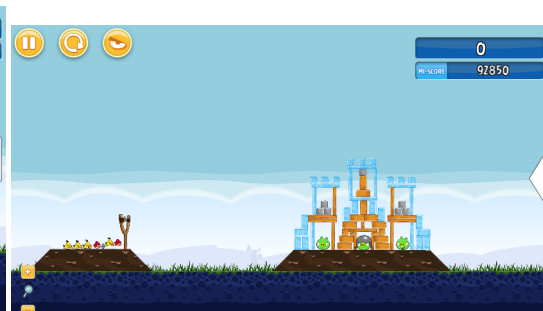


Figure 10.21: Poached Eggs : level 21

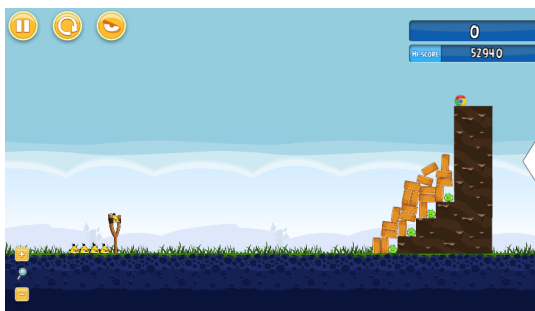


Figure 10.18: Poached Eggs : level 18

RectangleAlgebra.java

```
package ucl.utils;

import ab.vision.ABObject;

public class RectangleAlgebra {

    public static boolean left(ABObject x, ABObject y){
        if(x.x < y.x)
            return true;
        return false;
    }

    public static boolean before(ABObject x, ABObject y){
        if(x.getX()+x.getWidth() < y.getX())
            return true;
        return false;
    }

    public static boolean m(ABObject x, ABObject y){
        if(x.getX()+x.getWidth() == y.getX()){
            return true;
        }
        return false;
    }

    public static boolean mi(ABObject y, ABObject x){
        if(x.getX()+x.getWidth() == y.getX())
            return true;
        return false;
    }

    public static boolean mom(ABObject x, ABObject y){
        if((x.getX() < y.getX()) && (x.getX()+x.getWidth() >= ((y.getX()+y.
            getWidth())/2)) && (((x.getX()+x.getWidth())/2) >= y.getX()) && (x
            .getX()+x.getWidth() < y.getX()+y.getWidth()))
            return true;
        return false;
    }

    public static boolean momi(ABObject y, ABObject x){
        if((x.getX() < y.getX()) && (x.getX()+x.getWidth() >= ((y.getX()+y.
            getWidth())/2)) && (((x.getX()+x.getWidth())/2) >= y.getX()) && (x
            .getX()+x.getWidth() < y.getX()+y.getWidth()))
            return true;
        return false;
    }

    public static boolean lol(ABObject x, ABObject y){
```

```

        if (((x.getX()+x.getWidth())/2) < y.getX()) && ((x.getX()+x.getWidth()
            ) >= ((y.getX()+y.getWidth())/2)) && ((x.getX()+x.getWidth()) < (y
                .getX()+y.getWidth()))
            return false;
        return false;
    }

    public static boolean loli(ABObject y, ABObject x){
        if (((x.getX()+x.getWidth())/2) < y.getX()) && ((x.getX()+x.getWidth()
            ) >= ((y.getX()+y.getWidth())/2)) && ((x.getX()+x.getWidth()) < (y
                .getX()+y.getWidth()))
            return false;
        return false;
    }

    public static boolean mol(ABObject x, ABObject y){
        if ((x.getX() < y.getX()) && (x.getX()+x.getWidth() < ((y.getX()+y.
            getWidth())/2)) && (((x.getX()+x.getWidth())/2) >= y.getX()))
            return false;
        return false;
    }

    public static boolean moli(ABObject y, ABObject x){
        if ((x.getX() < y.getX()) && (x.getX()+x.getWidth() < ((y.getX()+y.
            getWidth())/2)) && (((x.getX()+x.getWidth())/2) >= y.getX()))
            return false;
        return false;
    }

    public static boolean lom(ABObject x, ABObject y){
        if (((x.getX()+x.getWidth())/2) < y.getX()) && (x.getX()+x.getWidth()
            >= ((y.getX()+y.getWidth())/2)) && ((x.getX()+x.getWidth()) < (y.
                getX()+y.getWidth()))
            return false;
        return false;
    }

    public static boolean lomi(ABObject y, ABObject x){
        if (((x.getX()+x.getWidth())/2) < y.getX()) && (x.getX()+x.getWidth()
            >= ((y.getX()+y.getWidth())/2)) && ((x.getX()+x.getWidth()) < (y.
                getX()+y.getWidth()))
            return false;
        return false;
    }
t }

    public static boolean ms(ABObject x, ABObject y){
        if ((x.getX() == y.getX()) && (x.getX()+x.getWidth() >= ((y.getX()+y.
            getWidth())/2)))
            return true;
        return false;
    }

```

```

public static boolean msi(ABObject y, ABObject x){
    if((x.getX() == y.getX()) && (x.getX()+x.getWidth() >= ((y.getX()+y.
        getWidth())/2)))
        return true;
    return false;
}

public static boolean ls(ABObject x, ABObject y){
    if((x.getX() == y.getX()) && (x.getX()+x.getWidth() < ((y.getX()+y.
        getWidth())/2)) && (x.getX()+x.getWidth() > y.getX()))
        return true;
    return false;
}

public static boolean lsi(ABObject y, ABObject x){
    if((x.getX() == y.getX()) && (x.getX()+x.getWidth() < ((y.getX()+y.
        getWidth())/2)) && (x.getX()+x.getWidth() > y.getX()))
        return true;
    return false;
}

public static boolean ld(ABObject x, ABObject y){
    if((x.getX() > y.getX()) && (x.getX()+x.getWidth() <= ((y.getX()+y.
        getWidth())/2)))
        return true;
    return false;
}

public static boolean ldi(ABObject y, ABObject x){
    if((x.getX() > y.getX()) && (x.getX()+x.getWidth() <= ((y.getX()+y.
        getWidth())/2)))
        return true;
    return false;
}

public static boolean mfi(ABObject x, ABObject y){
    if((x.getX() > y.getX()) && (x.getX() <= ((y.getX()+y.getWidth())/2))
        && (x.getX()+x.getWidth() == y.getX()+y.getWidth()))
        return true;
    return false;
}

public static boolean mfi(ABObject y, ABObject x){
    if((x.getX() > y.getX()) && (x.getX() <= ((y.getX()+y.getWidth())/2))
        && (x.getX()+x.getWidth() == y.getX()+y.getWidth()))
        return true;
    return false;
}

public static boolean lf(ABObject x, ABObject y){

```

```

        if((x.getX() > ((y.getX()+y.getWidth())/2)) && (x.getX() < (y.getX()+y.
            getWidth())) && (x.getX()+x.getWidth() == y.getX()+y.getWidth()))
            return true;
        return false;
    }

    public static boolean lfi(ABObject y, ABObject x){
        if((x.getX() > ((y.getX()+y.getWidth())/2)) && (x.getX() < (y.getX()+y.
            getWidth())) && (x.getX()+x.getWidth() == y.getX()+y.getWidth()))
            return true;
        return false;
    }

    public static boolean rd(ABObject x, ABObject y){
        if((x.getX() >= ((y.getX()+y.getWidth())/2)) && (x.getX()+x.getWidth()
            < y.getX()+y.getWidth()))
            return true;
        return false;
    }

    public static boolean rdi(ABObject y, ABObject x){
        if((x.getX() >= ((y.getX()+y.getWidth())/2)) && (x.getX()+x.getWidth()
            < y.getX()+y.getWidth()))
            return true;
        return false;
    }

    public static boolean cd(ABObject x, ABObject y){
        if((x.getX() > y.getX()) && (x.getX() < ((y.getX()+y.getWidth())/2))
            && ((x.getX()+x.getWidth()) > ((y.getX()+y.getWidth())/2)) && ((x.
            getX()+x.getWidth()) < (y.getX()+y.getWidth()))
            return true;
        return false;
    }

    public static boolean cdi(ABObject y, ABObject x){
        if((x.getX() > y.getX()) && (x.getX() < ((y.getX()+y.getWidth())/2))
            && ((x.getX()+x.getWidth()) > ((y.getX()+y.getWidth())/2)) && ((x.
            getX()+x.getWidth()) < (y.getX()+y.getWidth()))
            return true;
        return false;
    }

    public static boolean equal(ABObject x, ABObject y){
        if((x.getX() == y.getY()) && (x.getX()+x.getWidth() == y.getX()+y.
            getWidth()))
            return true;
        return false;
    }
}

```

UCLUtils.java

```
package ucl.utils;

import java.util.LinkedList;
import java.util.List;

import ab.vision.ABObject;

public class UCLUtils {

    /**
     * Return the list of ABObject that supports o1 or an empty list
     *
     * @param o1
     * @param objs
     * @return list of ABObject
     */
    public static List<ABObject> getSupporters(ABObject o1, List<ABObject>
        objs){
        List<ABObject> supporters = new LinkedList<ABObject>();

        for(ABObject o: objs){
            if(o1.isSupportBy(o)){
                supporters.add(o);
            }
        }

        return supporters;
    }
}
```

