Angry-HEX: an Artificial Player for Angry Birds Based on Declarative Knowledge Bases

Francesco Calimeri[1], Michael Fink[2], Stefano Germano[1], Andreas Humenberger[2], Giovambattista Ianni[1], Christoph Redl[2], Daria Stepanova[2], Andrea Tucci[1], and Anton Wimmer[2]

[1]Dipartimento di Matematica e Informatica, Università della Calabria, Italy
[2]Institut für Informationssysteme, Technische Universität Wien

This paper presents the Angry-HEX artificial intelligent agent that participated in the 2013 and 2014 Angry Birds Artificial Intelligence Competitions. The agent has been developed in the context of a joint project between the University of Calabria (UniCal) and the Vienna University of Technology (TU Vienna).

The specific issues that arise when introducing artificial intelligence in a physics-based game are are dealt with a combination of traditional imperative programming and declarative programming, used for modelling discrete knowledge about the game and the current situation. In particular, we make use of HEX programs, which are an extension of Answer Set Programming (ASP) programs towards integration of external computation sources, such as 2-D physics simulation tools.

***Index Terms*—Declarative Programming, Answer Set Programming, Artificial Intelligence, Computational Logic, Knowledge Representation and Reasoning, Physics-Based Games, Angry Birds, Competition**

I. Introduction

Angry Birds is a popular video game: its main goal is to destroy all pigs in a scene by shooting birds of different kinds at them using a slingshot. Score is given to the player according to the number of destroyed pigs and objects plus a bonus for each spared bird. The pigs are sheltered by complex structures made of objects of different materials (wood, ice, stone, etc.) and shapes, mostly but not exclusively rectangular. After a player's shot, the scenario evolves complying with laws of physics, with the crash of object structures and a generally complex interaction of subsequent falls.

Interestingly, in this game one can find many of the challenges that physics-based games present to the AI community. These are mostly related to the need of dealing with uncertainty in several respects, such as predicting the unknown consequences of a possible move/shot, or estimating the advantages of a choice with respect to possible alternative moves, or planning over multiple moves where any intermediate move is subject to failure or unexpected outcome. In turn, the above technical challenges require the effective resolution of other important issues, like the identification of objects via artificial vision, the combination of simulation and decision making, the modelling of game knowledge, and the correct execution of planned moves.

The Angry Birds AI Competition runs on a client/server architecture with one server and an instance of the browser-based version of the game for each participating agent. Each participating agent runs on a client which connects the browser game to the server according to a given protocol. An artificial player can also obtain the current reference scores for each level, and can prompt the server for executing a shot, which will in turn be performed in the corresponding game screen. The long-term goal of the competition series is to foster the building of AI agents that can play any previously unseen level better than the best human players. In order to successfully win this challenge, participants are solicited to combine different areas of AI such as computer vision, knowledge representation and reasoning, planning, heuristic search, and machine learning. Successfully integrating methods from these areas is indeed one of the great challenges of AI.

In this work, we present Angry-HEX, an artificial player for Angry Birds, based on declarative knowledge bases, that participated in the 2013 and 2014 Angry Birds AI Competitions [1], [2], which recently inspired a number of research contributions [3]–[6]. The agent features a combination of traditional imperative programming and declarative programming that allows us to achieve high flexibility in strategy design and knowledge modelling. In particular, we make use of Answer Set Programming (ASP) [7]–[12], a well-established paradigm of declarative programming for knowledge representation and reasoning (KRR), and HEX programs [13], which are a proper extension of ASP programs towards integration of external computation and knowledge sources; knowledge bases, written in ASP, drive both decisions about which target to hit for each shot (tactic gameplay), and which level should be played and how (strategic gameplay). The usage of Answer Set Programming has several benefits. First of all, ASP knowledge bases are much more flexible and easier to change than a procedural algorithm. In particular, thanks to the availability of constructs such as aggregate atoms and weak constraints, reasoning strategies can be easily encoded as a set of few rules, while the development of dedicated algorithms is time consuming and error prone. Therefore, conducting experiments with different strategies in order to optimize our agent is much easier. Furthermore, ASP is a general purpose declarative language in which temporal and spatial reasoning can be embedded as shown for instance in [14], [15]; also, ASP can be used in the planning domain, for modelling action languages (see e.g. the seminal work [16]), and probabilistic reasoning [17].

The advantages above come however at the price of a lower scalability when large input datasets have to be dealt with. In the context of the Angry Birds game, it is also necessary to deal with data coming from (approximately) continuous domains: Indeed, physics simulation libraries, using floating point operations, are needed; such data cannot be efficiently dealt with in a natively discrete, logic-based framework such as ASP. In this respect, HEX-programs, an extension of ASP, are well-suited as they allow for encapsulating external sources of information: on the one hand, numeric computations and part of the spatial processing tasks are computed on what can be called "the physics side" and can be embedded as external sources; on the other hand, actual reasoning strategies can be specified declaratively on a declarative "reasoning side". This way, numeric processing is hidden in the external sources, since returned elaborated results are limited to those aspects which are relevant for the declarative part; this can be encoded by a fairly limited set of discrete logic assertions.

The main contributions of the present work can be summarized as follows. Towards modelling dynamic (situation-dependent) and static knowledge in a physics-based game, we propose a hybrid model in which logic-based, declarative knowledge modelling is combined with traditional programming modules whose purpose is acting on the game and extracting discrete knowledge from the game itself. In our architecture, the decision support side is implemented in an extension of Answer Set Programming (ASP) integrated with external sources of computation modelled by means of traditional imperative programming. We contextualized this approach to the Angry Birds Game, and propose an agent which participated in the Angry Birds AI Competition Series. Also, we analyzed the performance of the proposed agent in several respects.

We point out next the advantages of the herein proposed approach. (i) It is possible to deal with the respective limitations of both declarative modelling and traditional programming and gain instead from respective benefits; indeed, on the one hand, logic programming is extremely handy and performance efficient when dealing with discrete domains, but it has limited ability to cope with nearly-continuous domains, and at the price of unacceptable performance. On the other hand, ad-hoc programmed modules lack flexibility but allow efficient processing of nearly-continuous knowledge (ballistic and geometric formulas, artificial vision, etc.). (ii) The introduction of declarative logic-based knowledge bases allows to combine statements concerning commonsense knowledge of the game (e.g., in the context of Angry Birds, "Blue birds are good on ice blocks") with objective knowledge (e.g. "an ice block w pixels wide is currently at coordinates (x, y)") when performing decision making; it permits also to focus attention on declarative descriptions of the game knowledge and of the goals to be achieved, rather than on how to implement underlying evaluation algorithms. This allows fast prototyping, and consequently much greater efficiency in the usage of developer time. For instance, both strategy and tactics behaviors can be easily refined and/or redefined by quickly changing logic assertions. (iii) Benchmarks show that logic-based approaches, and particularly, ASP-based approaches, if properly combined with procedural facilities, can be employed in applications having requirements near to real-time, making the gap between the performance of current logic-based solutions and requirements of pure real-time applications much narrower. (iv) The approach generalizes to a wide range of applications which share a number of aspects with the Angry Birds setting, such as automated room cleaning, semi-interactive turn-based games, planning in slowly evolving environments, such as robot and space probes etc. One can adapt our approach to such or similar scenarios, by adding a *process* step to the traditional *observe-think-act* cycle [18], thus obtaining a sort of *observe-process-think-act* cycle, and properly implementing the four stages. More in detail, the observation and process phases can be implemented by hardwiring sensors and processing their inputs in a procedural environment, providing a focused set of external sources of knowledge to the think phase; the process phase plays the role of discretizing and reducing information, so that the think phase, carried out by a logic-based system, is efficiently performed, still keeping the benefits of declarative knowledge modelling; the act phase is straightforward and rewires decisions to actual actions. A deployment of an approach similar to present work can be found in [19].

In the following, we first describe the Angry Birds AI competition setting; then we overview the agent architecture and outline specific design choices introduced for dealing with physics and uncertainty; then, we comment experimental results in terms of time and score performance. Finally, we draw conclusions and discuss open issues and future development.

II. The Angry Birds AI Competition Setting

We briefly describe the game and the competition setting.

A. Game Environment

For each agent participating in the competition, a unique corresponding Angry Birds game instance runs on a game server, while the agent itself is executed on a client computer. The competition machinery supports Java, C/C++ and Python agents, and can run the artificial players under either Windows or Linux. Each agent is allowed a total of 100MB of local disk space on the client computer, for its convenience, including the space required for the agent code. Client computers have no access to the internet, and can only communicate with the game server by means of a specific communication API. No communication with other agents is possible, and each agent can only access files in its own directory.

The communication with the game server allows each agent to obtain screenshots of the current game state to submit actions and other commands. The actual game is played over the Google Chrome (browser) version of Angry Birds, in SD (low-resolution) mode, and all screenshots have a resolution of 840×480 pixels.

B. Game Objects

The objects an agent might have to deal with correspond to all block and bird types, background, terrain, etc. occurring in the first 21 levels of the "Poached Eggs" level set available

at chrome.angrybirds.com. In addition, the competition levels may include white birds, black birds, so-called TNT boxes, triangular blocks and hollow blocks (triangle and squares).

Once a bird is shot, the player can perform an additional "tap" anytime afterwards, provided that it is performed before the bird touches any other object. This action causes different events according to each bird type: blue birds generate multiple little blue birds; yellow birds accelerate and become very good at breaking wood; white birds drop an explosive egg while accelerating towards the sky; black birds explode making great damage, and so on. "Tapping" at the appropriate moment in time can make shot outcomes vary greatly.

C. Game Levels and Competition Setting

The levels used in the competition are not known in advance to the participants and are not present in the original version of the game; throughout the competition, each game level can be accessed, played and re-played in arbitrary order by the agent. Participants have a total time budget to solve the competition levels corresponding to a few minutes per game level, on average. As an example, as reported by the official competition rules, for 10 levels there is a maximum allowed time of 30 minutes. Once the overall time limit is reached, the connection of agents with the game server is terminated, then the agents have up to two minutes to store any needed information and then stop running.

A sample agent (the so-called Naive agent) is launched on all qualification game levels in advance; it does not participate in the competition, but its high scores are recorded and intended to provide participants with a reference baseline. The Naive agent is programmed to randomly aiming at pigs, no matter of their shelters, with no particular tactics.

Some strategy is needed when the agent takes part in multiple rounds. In particular, each participating agent must be able to distinguish between qualification round 1, qualification round 2, and the Finals.

Qualifications are run in two rounds, both on the same level set. During the first qualification round, agents can obtain, besides their own scores, the per level high score obtained by the Naive agent; during the second qualification round, agents can obtain the overall per level high score obtained in the first round (among all participants). Agents can, for example, program the strategy by determining the game levels where they can obtain the highest improvements between the two rounds. Agents cannot be modified between round 1 and round 2.

The highest scoring agents after qualification round 2 participate to the finals, where they are divided into groups. During finals, any agent can query the current group high score for each game level (but not the high scores of other groups). Grand finals are played on groups of two agents only.

III. ASP and HEX Programs

The Angry-HEX agent models its knowledge of the game by means of a variant of ASP knowledge bases called HEX programs, and reasons on top of it via an appropriate solver. In this section we introduce both ASP and HEX programs.

A. Answer Set Programming

Answer Set Programming (ASP) [7]–[11], [20] is a well-established declarative programming approach to knowledge representation and reasoning, proposed in the area of non-monotonic reasoning and logic programming. ASP has a close relationship to other formalisms such as propositional satisfiability (SAT) [21], Satisfiability Modulo Theories (SMT) [22], Constraint Handling Rules (CHR) [23], PDDL (planning) [24], and many others. The fully declarative nature of ASP allows one to encode a large variety of problems by means of simple and elegant logic programs. The idea is to model a given problem domain and contingent input data with a knowledge base *KB* composed of logic assertions, such that the logic models (*answer sets*) of *KB* correspond to solutions of an input scenario; an ASP knowledge base might have none, one or many answer sets, depending on the problem and the instance at hand.

For instance, let us consider 3-colorability, a well-known NP-complete problem. Given a graph G, the problem is to decide whether there exists an assignment of one out of three colors (say, red, green, or blue) to each node of G such that adjacent nodes always have different colors. Suppose that G is represented by a set of assertions F using a unary predicate *node* and a binary predicate *arc*. Then, the following ASP knowledge base, composed of rules r_1 and r_2, declaratively describes 3-colorings:

$$r_1: \quad col(X, red) \mid col(X, green) \mid col(X, blue) \leftarrow node(X).$$
$$r_2: \quad \leftarrow col(X_1, C), col(X_2, C), arc(X_1, X_2).$$

Intuitively, a rule can be seen as a universally quantified first order statement. Rule r_1 expresses that each node must either be colored red, green, or blue;[1] rule r_2 has empty consequence, and is also called *integrity constraint*: roughly, it is not permitted that all the literals in the constraint body are true at the same time. Hence, r_2 triggers inconsistency if two adjacent nodes share the same color, thus excluding logic models encoding invalid colorings. Thus, there is a one-to-one correspondence between the solutions of the 3-colorability problem and the answer sets of $F \cup \{r_1, r_2\}$, and G is 3-colorable if and only if $F \cup \{r_1, r_2\}$ has some answer set.

ASP is nowadays employed in a variety of applications, ranging from classical AI to real-world and industrial applications, e.g. [25]–[27]. The needs addressed by such a variety of applications fostered a thriving research within the community, causing both the enrichment and standardization of the language (the language standard ASP-Core is nowadays at its 2.0 version [28]) and the development of efficient solvers (for a list, we refer the reader to [29]). Advancements are periodically assessed in the customary ASP Competition [29], whose goal is to assess the state of the art in ASP solving on challenging benchmarks.

[1] Variable names start with an upper case letter and constants start with a lower case letter. Note that, differently from first-order models, answer sets are required to be minimal: i.e. r_1 does not classically implies that a node cannot be assigned to more than one color, although the consequence of r_1 implicitly represent an exclusive disjunction in ASP.

B. HEX *Programs*

One can see that the main benefit of the introduction of a paradigm like ASP consists in the possibility of describing problem domains at a high abstraction level, rather than implementing specifically tailored algorithms. The ease of modelling comes at the price of evaluation performance (nonetheless, efficient ASP solvers are nowadays available, see [29]). Discrete logic-based modelling paradigms are however historically weak on *a)* modelling over continuous or nearly-continuous values, and have a limited capability of *b)* dealing with probabilistic or fuzzy values. Both aspects play a significant role in physics-based games, in which moving objects are modelled using floating point values, and outcomes of the game are subject to uncertainty.

In order to properly deal with such issues, we opted for making use of HEX programs. HEX programs are an extension of ASP which allows the integration of external information sources, and which are particularly well-suited when some knowledge of the problem domain at hand is better modelled with means other than discrete logic. We formally overview HEX programs next.

A signature consists of mutually disjoint sets $\mathcal{P}$ of predicates, $\mathcal{E}$ of external predicates, $\mathcal{C}$ of constants, and $\mathcal{V}$ of variables. $\mathcal{C}$ may contain constants that do not occur explicitly in a HEX program and can even be infinite.

A *(signed) ground literal* is a positive or a negative formula $\mathbf{T}a$ resp. $\mathbf{F}a$, where a is a ground atom of form $p(c_1, \ldots, c_\ell)$, with predicate $p \in \mathcal{P}$ and constants $c_1, \ldots, c_\ell \in \mathcal{C}$, abbreviated $p(\vec{c})$. An *assignment* I is a consistent set of literals. We make the convention that if an assignment does not explicitly contain $\mathbf{T}a$ or $\mathbf{F}a$ for some atom a, i.e. the assignment is partial, then a is false wrt. I. An *interpretation* is a complete assignment I, i.e., for every atom a either $\mathbf{T}a \in I$ or $\mathbf{F}a \in I$ holds.

1) Syntax

HEX programs generalize (disjunctive) extended logic programs under the answer set semantics [10] with *external atoms* of form $\&g[\vec{X}](\vec{Y})$, where $\&g \in \mathcal{E}$, $\vec{X} = X_1, \ldots, X_\ell$ and each $X_i \in \mathcal{P} \cup \mathcal{C} \cup \mathcal{V}$ is an *input parameter*, and $\vec{Y} = Y_1, \ldots, Y_k$ and each $Y_i \in \mathcal{C} \cup \mathcal{V}$ is an *output term*.

Each $p \in \mathcal{P}$ has arity $ar(p) \geq 0$ and each $\&g \in \mathcal{E}$ has input arity $ar_{\text{I}}(\&g) \geq 0$ and output arity $ar_{\text{O}}(\&g) \geq 0$. Each input argument i of $\&g$ ($1 \leq i \leq ar_{\text{I}}(\&g)$) has *type* **const** or **pred**, denoted $\tau(\&g, i)$, where $\tau(\&g, i) = \mathbf{pred}$ if $X_i \in \mathcal{P}$ and $\tau(\&g, i) = \mathbf{const}$ otherwise.

A HEX program consists of a set of assertions (rules) r of form

$$a_1 \mid \cdots \mid a_k \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n. \quad (1)$$

where each a_i is an (ordinary) atom and each b_j is either an ordinary atom or an external atom, and $k + n > 0$.

The consequence (*head*) of r is $H(r) = \{a_1, \ldots, a_n\}$, the premise (*body*) is $B(r) = B^+(r) \cup \text{not } B^-$, where $B^+(r) = \{b_1, \ldots, b_m\}$ is the *positive body*, $B^-(r) = \{b_{m+1}, \ldots, b_n\}$ is the *negative body*. For any rule, set of rules O, etc., let $A(O)$ and $EA(O)$ be the set of all ordinary and external atoms occurring in O, respectively.

a) Example:

the following rule exemplifies how external information is dealt together with classical logic assertions:

$$instability(O, I) \leftarrow \&stability[W, H](S), \\ object(O, W, H), I = 100 - S.$$

The above rule specifies the instability of an object given its *stability*. This latter value is computed externally by means of the atom $\&stability[W, H](S)$, which intuitively processes an object of width W and height H, returning its stability S, a value ranging in the interval $\{0, \ldots, 100\}$, describing a measure of how an object is prone to fall in the current scene.

2) Semantics

The semantics of a HEX program P is defined via its grounding $grnd(P)$ (over $\mathcal{C}$), where the value of a ground external atom $\&g[\vec{p}](\vec{c})$ wrt. an interpretation I is given by the value $f_{\&g}(I, \vec{p}, \vec{c})$ of a $k{+}l{+}1$-ary Boolean *oracle function* $f_{\&g}$ [13]. The notion of *satisfaction* of (sets of) ground literals, rules, programs etc. O wrt. an interpretation I (denoted $I \models O$, i.e., I is a *model* of O) extends from ordinary logic programs to HEX programs, by taking external atoms into account. That is, for every ordinary atom a, $I \models a$ if $\mathbf{T}a \in I$, and $I \not\models a$ if $\mathbf{F}a \in I$, and for every external atom $a = \&g[\vec{p}](\vec{c})$, $I \models a$ if $f_{\&g}(I, \vec{p}, \vec{c}) = 1$. For a rule r of form (1), $I \models r$ if either $I \models a_i$ for some $1 \leq i \leq k$, $I \models b_j$ for some $m < j \leq n$, or $I \not\models b_j$ for some $1 \leq j \leq m$. Finally, $I \models P$, if $I \models r$ for every $r \in P$. An *answer set* of a HEX program P is any model I of the *FLP-reduct* P^I of P wrt. I, given by $P^I = \{r \in grnd(P) \mid I \models B(r)\}$ [30], whose positive part $\{\mathbf{T}a \in I\}$ is subset-minimal, i.e., there exists no model I' of P^I such that $\{\mathbf{T}a \in I'\} \subset \{\mathbf{T}a \in I\}$.[2] The set of all answer sets of P is denoted by $\mathcal{AS}(P)$. Answer sets can be ranked according to cost functions, which are expressed by declarative optimization statements called *weak constraints* [31], [32]. Weak constraints express desiderata which *should* be satisfied. Let us consider again the 3-coloring example of section III-A), and imagine that we want to limit the number blue-colored nodes; then, we might add the rule $\{:\sim col(X, blue).\}$ Weak constraints might also feature weights and levels for a more thorough expression of optimization problems; for more details, we refer the reader to the literature.

IV. The Angry-HEX Agent

Since logic-based reasoning is not specifically tailored to reasoning with non-discrete domains, it is particularly challenging to deal with physics-based simulations. This technical challenge can be coped with a hybrid system. Hence, we propose a double-sided architecture, in which a "decision making" side and a "simulation side" can be identified. The decision support side is realized using a logic-based knowledge base, while the simulation side is out-sourced to specialized library code.

In particular, in the Angry-HEX Agent the decision making process is carried out by computing the answer sets of a number of HEX programs. Namely, the program P_{Tact} models

[2] The FLP-reduct is equivalent to the traditional Gelfond-Lifschitz reduct for ordinary logic programs [10], but more attractive for extensions such as aggregates or external atoms.

the knowledge of the game within a single level, i.e. tactics aspects; and P_{Strat} models the strategical knowledge required when deciding which level is convenient to be played. When decisions have to be made, both P_{Tact} and P_{Strat} are coupled with respective sets of logical assertions A_{Tact} and A_{Strat}, where A_{Tact} describes the situation in the currently played level, and A_{Strat} describes the overall game status (scores, etc.) Respectively, each answer set of $P_{Tact} \cup A_{Tact}$ describes a possible target object to be hit with a bird, while the answer sets of $P_{Strat} \cup A_{Strat}$ describe which is the next level to played.

It is worth mentioning that we did not make use of machine learning techniques or other means for automatically obtaining knowledge of the game. Tactics and strategy have been modelled based on our own experience and experiments with the game. The contribution of this paper is indeed focused on the ease of knowledge modelling rather than automated learning, which can be subject of future research.

A. Framework Architecture

The Framework architecture consists of several components as shown in Fig. 1. The *Angry Birds Extension* works on top of the Google Chrome browser, and allows to interact with the game by offering a number of functionalities, such as capturing the game window and executing actions (e.g., moving the mouse, clicking, zooming). The *Vision Module* segments images and recognizes the minimum bounding rectangles of essential objects, their orientation, shape and type. Objects include birds of various colors, pigs, the slingshot, and bricks made of several materials and shapes. The *Trajectory Module* estimates the parabolic trajectory that a bird would follow, given a particular release point of the slingshot. The *AI Agent* stub is supposed to include the artificial intelligence programmed by participants of the competition, therefore it is the core module implementing the decision making process. The *Game Server* interacts with the *Angry Birds Extension* via the *Proxy* module, which can handle commands like `CLICK` (left click of the mouse), `DRAG` (drag the cursor from one place to another), `MOUSEWHEEL` (scroll the mouse wheel), and `SCREENSHOT` (capture the current game window). There are many categories of messages (Configuration messages, Query messages, In-Game action messages and Level selection messages); the *Server/Client Communication Port* receives messages from agents and sends back feedback after the server executed the actions asked by the messages.

B. Other Improvements to the Framework Architecture

The framework utilities allow an agent to gather the information needed to play the game levels; hence, we enhanced some modules of the base architecture in order to fit our needs. These parts are reported in Figure 1 as boxes filled with slanted lines; in the following we discuss such improvements. (i) Concerning the Vision Module, we added the possibility of recognizing the orientation of blocks and the level terrain; even though these features were later implemented in the Framework Architecture by the competition organizers, we preferred to stick to our version, for what some particular

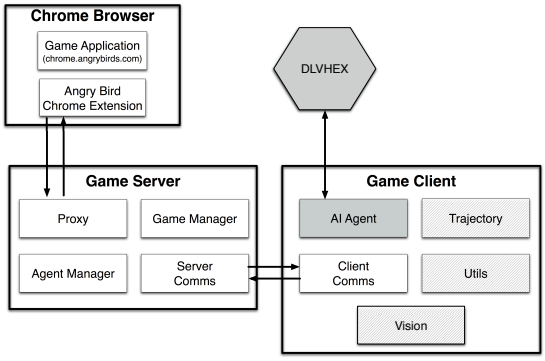


Fig. 1. The Framework Architecture. Slanted lines rectangles represents parts of the framework modified by our team, while gray ones represent the modules entirely developed by our team. Remaining modules were provided by the Competition organizers.

vision tasks are concerned. (ii) In the Trajectory Module, we added thickness to trajectories. A parabola is attributed a *thickness* value proportional to the size of a bird: this feature is helpful in order to exclude actually unreachable targets from the set of possible hits, because of narrow passages and sharp edges in objects' structures. Also, while the original Trajectory Module is capable of aiming at objects' centroids only, we can aim at several points taken on the left and on the top face of objects (recall that birds are always shot from the left-hand side of the screen). This has a two-fold benefit: first, objects that have their centroid hidden by other objects are not necessarily out of a possible hit, for instance, an object can have its top face clearly reachable while its centroid point is not; second, we can better choose the most convenient among a set of hitting points. For instance, higher points on the left face of an object are preferable because of an expectedly greater domino effect. (iii) Waiting for the outcome of a shot can be a time-consuming task: we added a quick-shot modality in which the next shot is performed after a fixed time, although there might still be slightly moving objects.

C. Our Agent

As already said, the core component of the framework Game Client is the AI agent. Figure 2 shows an overview of the Angry-HEX agent: the bot is composed of two main modules, the **Reasoner** and the **Memory**.

The memory module provides the agent with some learning mechanisms; in its current version, its first goal is to avoid that Angry-HEX replays the same level in the same way twice (for instance, by selecting a different initial target at the beginning of a level). Such an approach results to be quite effective, since changing the order of objects to be shot (even just the first one) results in completely different outcomes in terms of level evolution, and hence in future targeting choices and possibly improved scores for the same level. The reasoner module is in charge of deciding which action to perform. This module features two different intelligence layers: the **Tactic** layer, which plans shots and steers all decisions about "how" to play a level, and the **Strategy** layer, which establishes in what order the levels have to be faced; this layer decides also

whether it is worth replaying, not necessarily in a consecutive attempt, the same level more than once.

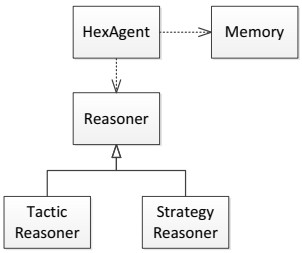


Fig. 2. An overview of the Angry-HEX Agent Architecture.

Tactic layer. The tactic layer is declaratively implemented using the DLVHEX solver, which computes optimal shots on the basis of the information about the current scene and the knowledge modeled within the HEX program P_{Tact}.

In particular, the tactic layer accesses and produces the following information. *Input data*: scene information encoded as a set of logic assertions A_{Tact} (position, size and orientation of pigs, ice, wood and stone blocks, slingshot, etc. as obtained by the Vision Module); a knowledge base P_{Tact} encoding knowledge about the gameplay. It is worth noting that physics simulation results and several other pieces of information are accessed within P_{Tact} via the so-called external atom construct. *Output data*: Answer sets of $P_{Tact} \cup A_{Tact}$ which contain a dedicated predicate *target* describing the object which has been chosen as a target and some information about the required shot, like the type of trajectory (high or low) and the hitting point (several points on the left and top face of the target can be aimed at).

We describe next the knowledge modeled by P_{Tact}. A shootable target T is defined as an object for which it exists a direct and unobstructed trajectory from the slingshot to T.

For each shootable target T, we define a measure of the estimated damage that can occur on all other objects if T is hit. The specific behavior of each bird type is taken into account (e.g. yellow birds are very effective on wood, etc.). Also, the estimated damage takes into account the probability of specific events. The higher the estimated damage function value, the better the target.

Targets are ranked by taking first those which maximize the estimated damage to pigs; estimated damage to other objects is taken into account, on a lower priority basis, only for targets which tie in the estimated damage for pigs.

The *optimal answer set*, containing, among its logical consequences the optimal target T_{opt}, is the answer set maximizing the damage function (see section V-A).

T_{opt} is then passed to the Trajectory Module. This latter module computes the actual ballistic data needed in order to hit T_{opt} (see Sub-section IV-B for more details).

Next we show some typical assertions used in Angry-HEX; for the sake of simplicity, we report a properly simplified version of the rules, even though the general idea is respected. In the following, with variable names of type $T[_i]$ and $O[_i]$ we refer to trajectory types and objects, respectively.

$$target(O,T) \mid nontgt(O,T) \leftarrow shootable(O,T). \ \Big\} \textbf{Guess}$$

$$\left.\begin{array}{l} \leftarrow\ target(O_1,\_),\ target(O_2,\_),\ O_1 \neq O_2. \\ \leftarrow\ target(\_,T_1),\ target(\_,T_2),\ T_1 \neq T_2. \\ target\_exists\ \leftarrow\ target(\_,\_). \\ \leftarrow\ not\ target\_exists. \end{array}\right\} \textbf{Check}$$

Intuitively, the first rule expresses that each shootable object can be possibly aimed, while the constraints (the "check" part) ensure that exactly one target is chosen.

Strategy layer. Upon completion of a level, the strategy layer decides which level should be played next. Like for the tactics layer we pursued a declarative ASP approach for the implementation of the strategy layer. This module is modelled by means of an ASP program, and the next level to be played is conveniently extracted from its logical consequences. This approach significantly improves the previous version of the Angry-HEX agent, where the strategy was hard-wired in Java. The ASP program P_{Strat} contains appropriate modelling of the following guidelines on the choice of the next level (here order reflects priority). (1) Play each level once. (2) Play levels for which the gap between our agent's score and the current best score is maximal (up to a limited number of attempts k). (3) Play levels where Angry-HEX outperforms all other agents, but its score minimally differs from the second best result (up to a certain number of attempts k'). (4) If none of the above rules is applicable, play a random level.

The program P_{Strat} has several pieces of input data available, reflecting the history of the game with respect to the played levels and scores achieved. Moreover, for each level, the strategy layer keeps track of previously selected target objects and, as mentioned, ensures the avoidance of repetition of the same opening shot on a particular level, thus allowing multiple approaches at solving the same level.

For example, the encoding of the first guideline in the ASP environment is provided in the following two rules:

$$\begin{array}{rrl} r_1: & chooselevel(1) \leftarrow & timeslevelplayed(1,0), \\ & & myscore(1,0). \\ r_2: & chooselevel(X) \leftarrow & timeslevelplayed(X,0), \\ & & timeslevelplayed(Y,Z), \\ & & myscore(X,0), \\ & & \#succ(Y,X), Z \geq 1. \end{array}$$

Rule r_1 schedules level 1 at the beginning of the game. The rule r_2 states that if a level X has not been yet played (represented by predicates $timeslevelplayed(X,0)$ and $myscore(X,0)$), X comes next after Y (predicate $\#succ(Y,X)$), and Y has been played more than once ($timeslevelplayed(Y,Z)$, $Z \geq 1$), then we choose the level X as the next one to be scheduled ($chooselevel(X)$). For instance, if the facts $timeslevelplayed(4,0)$, $myscore(4,0)$, $timeslevelplayed(3,1)$, are available due to the rules described above, the fact $chooselevel(4)$ will be deducted, and then level 4 will be scheduled.

V. Reasoning with Physics-Based Simulation

The "simulation side" allows to access and manipulate information typically not tailored to being dealt with a logic-based decision support system; in particular, we employ external atoms constructs to perform physics simulations and spatial preprocessing that help us to decide where to shoot. This way, the actual physics simulation and the numeric computations are hidden in the external atoms. The external atoms summarize the results in a discrete form.

Given a current level state, external processing is used for several tasks such as: determine if an object is *stable* (i.e., prone to an easy fall); determine whether an object B will fall when object A falls due to a structural collapse, or if it can be *pushed* (i.e., A can make B fall by domino effect); determine which objects intersect with a given trajectory of a bird, and in which sequence; determine if an object O is *shootable* (i.e., there exist a trajectory with O as the first intersecting object); find the best trajectory for a White Bird (white birds have a peculiar behavior and require a special treatment).

In the following we present a more detailed description of the simulation information we used. The data coming from the simulation side is fed into the decision making side as input assertions and by means of external atoms. Input assertions approximately encode statical information which is available a priori, like the current position of objects. External atoms elaborate and produce information triggered by the decision making side, like running a physics simulation and providing its outcome in terms of the number of falling objects, etc.

Input assertions. The input assertions, in the form of logical facts, encode information about the position of objects in a scene and data needed for trajectory prediction, such as:

$birdType(BT)$. The type of bird that is currently on the slingshot.

$slingshot(X, Y, W, H)$. The size and position of the slingshot, used for trajectory prediction.

$velocity(X)$. The current velocity scale. This is a value used internally by the trajectory prediction module.

$object(O, M, X, Y, W, H, A)$. There is one assertion of this kind for each object in the scene. Objects are enumerated for unique identification with ID O. Material M can be any of $ice, wood, stone, pig, ground$. Location of centroid (X, Y) of the rotated rectangle denotes the position of the object, width W and height H denote its size, and angle A denotes the rotation of the object at hand.

External Atoms. The following information sources available to the HEX program are implemented as external atoms. All external atoms are implemented using the physics software Box2D, which has been chosen for being a well documented, supported and regularly updated 2D physics engine. Box2D is widely used for game development and indeed the same Angry Birds game uses the library. All objects ($objs$) are added to a 2D World that we will call W. A simulation is then started on W, and the world is allowed to settle (as shown in Fig. 5). Usually there are small gaps between the objects, because of vision inaccuracies. Gaps need to be explicitly dealt with, since it is desirable that an object should still be detected as resting on the other object even if there is a one pixel gap: we therefore let the physics software proceed up to equilibrium (i.e. until all the objects are at rest). After every time step of the simulation we set all directional and angular velocities to zero, to avoid objects gaining speed and crashing buildings, and in order to keep the settled world as close to the original as possible. Let W^* be the 2D scene computed from W as above. In the following we assume that the mentioned external atoms implicitly operate on W^*.

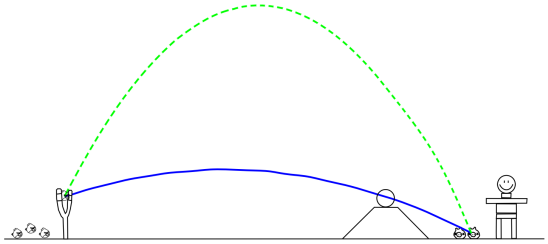


Fig. 3. An illustration of low and high trajectories (solid and dashed line respectively).

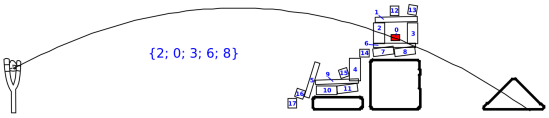


Fig. 4. An example of the output from the &next atom.

$\&on\_top\_all[objs](O_u, O_l)$. Allows to browse the set of couples of objects O_u, O_l for which O_u lies on top of object O_l. This information is used assuming that if O_l does not exist, O_u would likely fall. This is determined by checking whether object O_u exerts a force on O_l , that is oriented downwards. If so, we assume that O_u rests on O_l. In order to improve performance, a graph of object dependencies is calculated on the first call to this atom and, subsequently, cached answers are served.

$\&next[D, T_O, T_j, V, S_x, S_y, S_w, S_h, objs](I, O)$. For a bird trajectory aimed at object D, of type T_j, $\&next$ allows to inspect which objects are intersected by such a trajectory. T_j can either be $high$ or low (see Figure 3). V, S_x, S_y, S_w, S_h are helper variables required by the trajectory prediction module. V is the velocity scale, available from the $velocity(\dots)$ atom and S_x, S_y, S_w, S_h are the slingshot position and dimension values, available from the atom $slingshot(\dots)$. I is the position of the object O in the sequence of objects that would be hit in the trajectory T_j (e.g. in Figure 4 the object #2 has position 1 and so on). The offset T_O allows to choose from a set possible hitting points on the exposed faces of O.

$\&shootable[O, T_j, V, S_x, S_y, S_w, S_h, B, objs](O, S, U)$. This statement is true if object O is shootable with trajectory type T_j, i.e. if there exists a parabola whose O is the first intersected object from left to right. Most of the terms have the same meaning of ones in the $next$ atom. B identifies the bird type, which could be one of red, yellow, blue, black, white, for the thickness of a bird is used when determining shootability. S and U are the best offsets positions over O faces for the given trajectory and the given bird type.

(a) (b)

Fig. 5. An example of a level (#4 from the Theme One of the original set "Poached Eggs") and the corresponding reconstruction made by our external atoms using Box2D.

$\&firstbelow[P, objs](O)$. Denotes that the object with ID O is directly below the object P, with no items in between. We calculate this similarly to the $\&next$ atom, but instead of a trajectory we use a straight upward ray from O to P. The statement holds only if P is the first intersected object by such a ray.

$\&stability[W, H](S)$. Denotes the stability S of an object given its parameters W, H ($width$ and $height$ respectively). S is the ratio $(width/height) \times 50$ rounded to the nearest integer or 100 if it is greater than 100.

$\&canpush[objs](O_A, O_B)$. For each object O_A, selects all the objects O_B that can be pushed by O_A by a left-to-right domino fall. The canpush relation between an object O_A an O_B is computed by geometrically checking whether O_A, if rotated of 90 degrees rightwards, would overlap the O_B extension.

$\&clearsky[O, objs]$. This atom is specific for white birds and identifies whether the object O can be hit by the egg of a white bird. That is, it is determined whether there is enough space above O to let a White Bird to vertically release its egg on the object.

$\&bestwhite[O, T_j, V, S_x, S_y, S_w, S_h, objs](Y)$. Again, with specific focus on White Birds behavior, this atom returns the best height Y above the object O where to shoot, with trajectory type T_j, in order to achieve a good shot. A shoot with a White Bird on object O is considered optimal if it actually hits O and maximizes the damage effects of the "departure at 45 degrees" of the white bird in order to hit other objects. The other terms have the same meaning as in the $\&next$ atom.

The following are some examples of logic rules featuring external atoms; pushDamage intuitively describes the likelihood of damage when an object Obj_B is "pushed" by an adjacent object Obj_A:

$$\begin{aligned} pushDamage(Obj_B, P_A, P_B) \leftarrow & \\ & pushDamage(Obj_A, \_, P_A), P_A > 0, \\ & \&canpush[ngobject](Obj_A, Obj_B), \\ & pushability(Obj_B, Pu_B), \\ & P = P_A * Pu_B/100. \end{aligned} \tag{2}$$

$\&canpush$ works as described above, and allows to determine whether Obj_A can make Obj_B fall by means of a *domino effect*. It is worth noticing that $\&canpush$, as well as other atoms, uses geometric computations in a continuous space, however the values returned are discrete, in this particular case the result of the atom evaluation corresponds to the truth of a set of propositional atoms. P_B is a damage value estimate expressed as an integer value ranging from 0 to 100, and obtained as the product between the push damage P_A of Obj_A and the *pushability* Pu_B of Obj_B, normalized in the integer range $0, \ldots, 100$. The *pushability* value for an object is defined relying on empirical knowledge of the game, and defines how much an object can be *pushed* in terms of its shape, stability and material (e.g. long rods are easily pushable, etc.). Another example follows.

$$\begin{aligned} eggShootable(Obj, X) \leftarrow & \ \&clearsky[Obj, objects](), \\ & ngobject(Obj, \_, X, \_, \_, \_, \_). \end{aligned} \tag{3}$$

The above rule checks if an object that is not the scene ground surface ($ngobject$) can be hit by the egg released by a White Bird. Again, like in rule 2, the computation in the continuous space is entirely performed by the external atom.

A. The estimated damage function

The aim of the reasoning engine is to find the "best" object to shot, the most appropriate tap time and "how" the object should be hit (i.e., where to aim – to the center of the object, to a long or a short side, etc.). In order to identify the best target object, we attribute to each possible target a score Sc_1 based on the sum of damage likelihood for each pig and TNT box in the scene, and a score Sc_2 based on the sum of damages of other objects. We select the target that maximizes Sc_1, or, in case of a tie, we maximize Sc_2.

In turn, per each object, we attribute several damage type quotas. In general, all the damage types are computed in terms of causal event chains, in which damage is linearly weighted by the likelihood of the event causing the damage. The likelihood of an event is in turn obtained by the product of fixed empirical values combined with the likelihood of previous events in the causality chain. Damage types are described next.

direct damage: the damage an object takes when hit by a bird. Direct damage is calculated by examining the sequence of objects that intersect the assumed trajectory using the $\&next$ atom. This type of damage depends on the intrinsic *damage probability* P of each object and on the *energy loss* E of the bird (the farther an object is in the intersected object list the lesser its direct damage value). The following is an example of a rule to compute the direct damage:

$$\begin{aligned} directDamage(Obj, P, E) \leftarrow & \ target(Inner, Tr), \\ & next(Obj, 0, Outer, T, \_), \\ & objectType(Obj, T), birdType(Bird), \\ & damageProbability(Bird, T, P), \\ & energyLoss(Bird, T, E). \end{aligned} \tag{4}$$

The $next$ atom summarizes the external $\&next$ by means of rules like the following:

$$\begin{aligned} next(X, Y, Z, T, C) \leftarrow & \ shootable(X, T, C, \_), \\ & \&next[X, C, T, V, Sx, Sy, Sw, Sh, objects](Y, Z), \\ & velocity(V), slingshot(Sx, Sy, Sw, Sh), T \neq egg. \end{aligned} \tag{5}$$

In the above, the truth of an assertion $next(X, Y, Z, T, C)$ can be read as "X is the Y-th object in the trajectory T aiming at object Z, with horizontal shift C from the object's centroid".

push damage: the damage an object undergoes when pushed by another object. It is calculated by building the already mentioned chain of "pushes", and depends on the intrinsic *pushability* of an object and on the values coming from the external atom $\&canpush$. An example illustrating the role of &canpush is provided in rule (2) above.

fall damage/fall desirability: the damage an object undergoes when another object is destroyed below it. It is calculated according to the values of the $\&on\_top\_all$ external atom, and it depends on the intrinsic *material fall importance* P_N of the object and on all other kinds of damages (i.e. a fall damage chain can be started by a direct or push damage chain). For instance, in order to compute the damage of a "falling" object Obj one can specify the following rule:

$$\begin{aligned} &fallDamage(Obj, P) \leftarrow \\ &\quad pushDamage(RemovedObj, \_, P_R), P_R \geq 50, \\ &\quad \&on\_top\_all[objects](Obj, RemovedObj), \\ &\quad objectType(Obj, T), materialFallImportance(T, P_N), \\ &\quad P = P_R * P_N/100. \end{aligned} \tag{6}$$

B. Modelling empirical knowledge

A good portion of empirical knowledge of the gameplay is encoded in terms of logical assertions. As opposed to hard-wiring this information into traditional code, this allows better flexibility and easier fine-tuning and troubleshooting.

One type of such empirical knowledge comes into play when static object damage values are combined in order to form causal chains. Causal chains terminate using an energy loss estimate; energy losses take into account the residual energy available when the effects of a shot propagate from an object to another. Also, we model the attitude of a particular bird type towards destroying different material types. For instance, the following assertions encode the damage probability of an object depending on the object material and on the type of bird hitting the object at hand. They correspond to intuitive statements like "Blue birds are very good on ice blocks":

$$\begin{aligned} &damageProbability(blue, wood, 10). \\ &damageProbability(yellow, wood, 100). \\ &damageProbability(blue, ice, 100). \\ &damageProbability(yellow, ice, 10). \end{aligned}$$

Other empirical knowledge includes the following.

damage probability: for each couple *(Bird_Type, Material)* we encode the damage an object made of *Material* receive when hit by *Bird_Type*.

energy loss: for each couple *(Bird_Type, Material)*, we encode the reduction of energy a bird of *Bird_Type* experiences when it destroys an object made of *Material*.

pushability: it denotes how "easy" is an object made of a given material to be pushed, when other conditions and features (i.e., shape) are fixed. For instance, stones react less to pushes than wood.

material fall importance: the damage an object made of a given material can cause when it falls, under other conditions equal, like size. For instance, stones are assumed to have greater density.

A second group of empirical gameplay information comes into play when dealing with trajectory prediction. The Trajectory prediction module, given in input some target coordinates, considers several "high" (aiming at a vertical fall to an object) and several "low" trajectories (aiming at a horizontal hit on the left-hand face of an object), but returns only two of both categories. Many candidate trajectories are discarded because of obstructions before the target point. This discretization is done in order to reduce the space of possibilities which the reasoning module has to take decisions on. This approximation can be considered acceptable, and we indeed did not experimented appreciable differences in the effects of two different parabolas of same category.

Trajectory prediction is treated differently depending on the type of bird that is on the slingshot at reasoning time. For what red, yellow, blue and black birds are concerned, we use a normal "parabolic" trajectory. This trajectory aims at the objects' left face if a *low* trajectory is selected while the top face is aimed at if an *high* trajectory is chosen. Three hitting points are possible for each face, for a total of 12 possible trajectories per object.

As for tapping time, we tap yellow birds relatively close to the target object, so to maximize their acceleration effect without compromising the parabolic trajectory; blue birds are tapped a bit earlier in order to maximize their "spread" effect, while black birds are tapped right on target so to maximize their explosion effect.

The white bird receives special treatment: we first try to identify which objects can be hit by the vertical fall of the *egg* that a white bird can release with a tap. We then choose the best point where the bird should release the egg itself, in terms of side effects, since the white bird continues its trajectory after laying its egg, thus creating more damage chains.

VI. Results/Benchmarks

In this section we discuss experiments and performance.

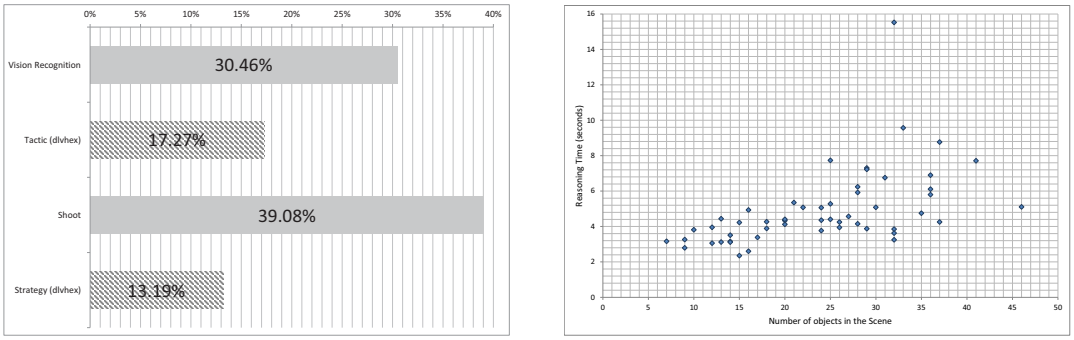
A. Competition outcomes and Third party benchmarks

Angry-HEX performed quite well in the 2013 and 2014 Angry Birds AI Competitions. Our team participated also in 2012, but with two preliminary agents, that were also largely different from the herein presented Angry-HEX agent. Our agent reached semifinals in 2013 (being the best one during all previous rounds) and quarterfinals in 2014.[3]

The Organizing Committee performed also some benchmarks over the participating agents in the same settings of the competition, in order to allow participants to easily compare the performance of their agents with others. The benchmarks were run on the first 21 levels of the freely available Poached Eggs levels, and each agent had a time budget of 63 minutes, corresponding to an average of 3 minutes per level. Table I of Supplementary Material[4] shows the results of the Benchmarks executed by the organizers. Teams with names in bold participated in the 2014 Competition, the others in the 2013 Competition. Bold numbers represent high-scores of each level among all participants.

[3]For official results see public data available at https://goo.gl/aP460U and https://goo.gl/V0gVse (Table II and III in Supplementary Material).

[4]Publicly available at https://goo.gl/y72QMP



(a) (b)

Fig. 6. (a) The average percentage of time spent in each "main task" by our agent. Solid bars account for tasks common to all the agents (i.e. tasks that are performed by using functionalities provided by the organizer's framework); Slanted lines bars account for the specific tasks of our agent, performed using our reasoning modules. (b) The time spent by the Tactic layer with respect to the number of objects in the scene.

In the after-show benchmarks of 2013, Angry-HEX performed better than all other participants. Our score was very good in many levels (reaching the 3 stars score), even if only in one level we had the best score among all the participants. In the after-show benchmarks of 2014, Angry-HEX performed better than most of the other participants that had outperformed it in the quarterfinals. In some levels we performed similarly to the 2013 (usually with a slightly lower score); however, in some other we performed much better. Similarly to what happened in the previous edition, in 2014 no agent was able to perform better than others in all the levels, and also this year the best scores of each level was done mostly by the agents in the center/bottom part of the classification. The fact that the scores in each level are typically distributed over many participants that are listed below Angry-HEX in rankings might hint that the strategies implemented in other agents are tailored to specific types of levels, but generalize less, or worse, than the one of Angry-HEX. A proper deeper analysis of the levels where Angry-HEX was not the best this year could be a fruitful approach for defining special cases in its tactics. It is worth to notice that the agents almost always reach a 3-star score, that represent a very good score to achieve for a human player. However, to date, the "Man vs Machine Challenge", in which each year human players compete with the winners of the Angry Birds AI Competitions, was always won by human players. Eventually, it is important to notice that the differences between our results in these 2 years are very little, and are mostly due to a large restructuring of the agent code, which has been performed in 2014 in order to make it more extensible, flexible and easy to install. Therefore, we expect to take advantage of this work in the forthcoming competition editions.

Remark. At the time of printing, we just received the results of the 2015 Competition; our agent ranked 2nd overall. Complete results are reported in the Supplementary Material, and confirm what discussed above.

B. Tests on time performance

The tactic layer is repeatedly prompted for deciding the chosen target on the current scene: reducing reasoning times is crucial in order to better exploit the allowed time and improving scores. We recall that the core of the tactic layer is an evaluation machinery carried over the logical knowledge base P_{Tact}, coupled with a set of assertions A_{Tact} that describe the current scene; hence, both the size of A_{Tact} and P_{Tact} affects its performance. Another important performance factor is the number and duration of calls to external libraries.

The size of A_{Tact} is directly proportional to the number of objects in the current scene: there is one logical statement for each object in the scene, plus a constant number of facts encoding the slingshot position, the current bird type, and the scale factor of the scene, respectively. P_{Tact}, instead, is a fixed knowledge base featuring about three hundred statements, made both of rules and facts that encode the domain knowledge. For what calls to external libraries are concerned, these were optimized with the aim of reducing the number of possibly redundant computations.[5]

Due to the unavailability to the public of the levels used in the official Angry birds AI Competition, our comparative studies of actual performance are limited to the publicly known first 21 levels of the "Poached Eggs" level set,[6] though they do not explicitly stress reasoning capabilities of artificial agents. The experiments were conducted on a virtual machine running Ubuntu 14.04.2 LTS, containing 2-cores of a 2.29 GHz Quad-core Processor and 3 GB of RAM, running standalone in its hypervisor. First, we noted that the reasoning time was a small fraction of the overall time spent for each shot; indeed, most of the time is spent by animations and the simulation of the shot itself, as shown in Fig. 6a.

Fig. 6b depicts the number of objects in the scene of a level against the time spent by the reasoning tasks within the Tactic Layer, on that level. It is worth noting that the number of objects in a level is not a direct measure of "how hard" is a level from a player perspective: this depends on the materials, the shapes, the (relative) positions, the way pigs are sheltered, and so on. The number of objects in a scene is however proportional to the size of data given in input to the

[5]Furthermore, the burden of repeated identical calls to external sources has been mitigated by caching.

[6]"Poached Eggs" is the first level set available in the publicly downloadable version of the game, which reflect the competition setting in qualitative terms (type of materials, shape of objects and type of birds available), and are customarily used by the organizers in the after-show benchmarks

Tactic Layer. We found some direct proportionality between time and the number of objects in a scene, but, interestingly, the system seems to scale fairly well with respect to the latter. The fraction of reasoning time dedicated to external atom calls did not show a clear trend: it averaged around 61% of the total reasoning time, with low and high peaks of 30% and 90% respectively.

We also focused in measuring the performance of the Tactic Layer when changing tactics knowledge bases. In order to compare the impact on time performance, when changing tactics, we measured the reasoning time of three, incrementally better in terms of gameplay, different reasoning knowledge bases. The cactus plot in Figure 7 shows the performance of the Tactic Layer (i.e. the time taken to take a decision given an input scene already processed by the Vision module), for all the runs needed for completing the Poached Eggs levels (54 shots in total); in order to give a clearer glimpse at the resulting trends the data series were sorted by execution time. We experimented with three different tactics: *directDamage* (a knowledge base of 133 logic assertions), in which only estimated damage by direct hits is maximized; *pushDamage* (143 assertions), in which we add the potential damage of domino effect, and *fallDamage* (155 assertions), which corresponds to the tactics participating to the Competition, in which we add also damage due by the vertical fall of objects. It is easy to see that the time differences are not substantial, especially if compared to the time cost of other evaluation quotas (vision, gameplay, etc.) with *fallDamage* clearly asking for a somewhat larger price in terms of time consumption. We can conclude that, when designing knowledge bases, a fairly large degree of freedom in designing sophisticated tactics can be enjoyed without being worried by a loss in time performance. External calls were a key-point for efficiency: the strategy of outsourcing several tasks outside the decision making core proved to be useful. As a matter of fact, the performance of one earlier attempt of implementing an ASP-only agent capable of playing a real-time game was far from satisfactory [33], given the fairly limited efficiency of ASP solvers; we believe that implementing this AI player using ASP (with no external calls and/or a "process" stage), if feasible at all, would not have fit with the real-time requirements of the game.

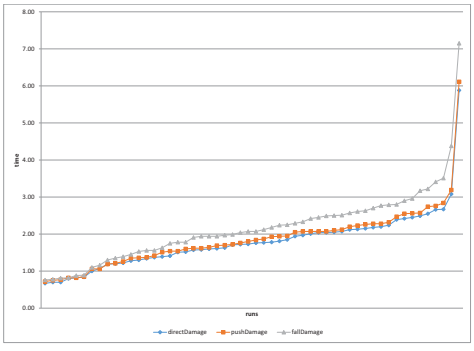


Fig. 7. Reasoning times for different tactics knowledge bases, measured on the first 21 levels of "Poached Eggs"

C. Impact of this work from the knowledge representation and reasoning perspective.

The setting of the Angry Birds game has two particular aspects: first, Angry Birds can be seen as a game which lies somehow between a real-time game and a turn-based one, allowing a fairly large time window for deciding the next shot; second, at each step, the tactic layer must explore a search tree whose size is reasonably small and polynomially proportional to the number of objects in the current scene.

The above setting can be lifted to a variety of other contexts in which *(a)* there is an acceptable time window available for reasoning, and *(b)* the set of actions that can be taken by the agent at hand is fairly small, do not underly an exponentially larger search space, and no look-ahead of arbitrary depth on future scenarios is required (or convenient) to be performed. This generalized context covers e.g. planning in slowly evolving environments (e.g. robot and space probes), automated room cleaning, semi-interactive turn-based games, etc. In this respect, the Angry-HEX agent constitutes a good proof-of-concept showing how ASP, if properly deployed in a hybrid architecture and extended with procedural aids, not only qualifies as an effective tool for implementing near real-time solutions while enjoying the advantages of declarative approaches, but witnesses that the realizability of real-time applications is much closer.

VII. Conclusion

The Angry Birds game, and hence the study, design and implementation of this work, led to face several challenges for knowledge representation and reasoning, and artificial intelligence in general; eventually, we can make some considerations. First of all, it looks clear that, in order to accomplish complex jobs/tasks, a monolithic approach should be discarded in favour of more diversified ones, consisting of convenient integrations of various, if not many, methods and technologies. In this respect, any bundle of KRR formalism/system of use, besides expressive power, suitable syntax/semantics and good efficiency/performance, must feature proper means for easing such integration at all levels, from within the language to the actual implementation. The work carried out by the scientific community in the latest years and the effective use of ASP in real-world and industry-level applications [34], suggest Answer Set Programming as a powerful tool in such scenarios; and the present work confirms this idea. Also performances do not constitute a big issue, as discussed in section VI-B.

As for the original goal of Angry-HEX, even though from the benchmarks and the results of the competitions our approach seems quite effective and general, we further identified several aspects in which Angry-HEX can be improved. Most importantly, we aim at introducing the planning of multiple shots based on the order of birds that must be shot (it is worth remembering that the type and number of birds, as well as the order the player have to shoot them, is given at the beginning of each level); we think that this might prove to be useful especially when dealing with complex levels. A more accurate study of the interaction between objects, and a more detailed implementation of the different shapes of the objects

are also under consideration. Furthermore, we aim to not affect reasoning time when introducing improvements.

ACKNOWLEDGMENT

Partially supported by MIUR under PON project "SI-LAB BA2KNOW Business Analitycs to Know", and by Regione Calabria, programme POR Calabria FESR 2007-2013, projects "ITravel PLUS" and "KnowRex: Un sistema per il riconoscimento e lestrazione di conoscenza".

The authors would like to thank the anonymous reviewers for their careful reading and the detailed and constructive comments, that allowed us to significantly improve the paper.

REFERENCES

[1] J. Renz, X. G. Ge, P. Zhang, and S. Gould, "Angry Birds AI Competition," http://aibirds.org, Australian National University, 2014.

[2] J. Renz, "AIBIRDS: the angry birds artificial intelligence competition," in *Proc. of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, Austin, Texas, USA, January 2015, pp. 4326–4327.

[3] P. Zhang and J. Renz, "Qualitative spatial representation and reasoning in angry birds: The extended rectangle algebra," in *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*, C. Baral, G. D. Giacomo, and T. Eiter, Eds. AAAI Press, 2014. [Online]. Available: http://www.aaai.org/ocs/index.php/KR/KR14/paper/view/8021

[4] A. Narayan-Chen, L. Xu, and J. Shavlik, "An empirical evaluation of machine learning approaches for angry birds," in *IJCAI Symposium on AI in Angry Birds*, 2013.

[5] N. Tziortziotis, G. Papagiannis, and K. Blekas, "A bayesian ensemble regression framework on the angry birds game," *arXiv preprint arXiv:1408.5265*, 2014.

[6] F. Calimeri, M. Fink, S. Germano, G. Ianni, C. Redl, and A. Wimmer, "Angryhex: an artificial player for angry birds based on declarative knowledge bases," in *Proc. of the Workshop PAI co-located with AI*IA 2013, Turin, Italy, December 5*, 2013, pp. 29–35.

[7] C. Baral, *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[8] T. Eiter, W. Faber, N. Leone, and G. Pfeifer, "Declarative Problem-Solving Using the DLV System," in *Logic-Based Artificial Intelligence*, J. Minker, Ed. Kluwer Academic Publishers, 2000, pp. 79–103.

[9] M. Gelfond and N. Leone, "Logic Programming and Knowledge Representation – the A-Prolog perspective ," *Artificial Intelligence*, vol. 138, no. 1–2, pp. 3–38, 2002.

[10] M. Gelfond and V. Lifschitz, "Classical Negation in Logic Programs and Disjunctive Databases," *New Generation Computing*, vol. 9, pp. 365–385, 1991.

[11] V. Lifschitz, "Answer Set Planning," in *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, D. D. Schreye, Ed. Las Cruces, New Mexico, USA: The MIT Press, Nov. 1999, pp. 23–37.

[12] M. Alviano, F. Calimeri, W. Faber, N. Leone, and S. Perri, "Unfounded sets and well-founded semantics of answer set programs with aggregates," *J. Artif. Intell. Res. (JAIR)*, vol. 42, pp. 487–527, 2011. [Online]. Available: http://dx.doi.org/10.1613/jair.3432

[13] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits, "A uniform integration of higher-order reasoning and external evaluations in answer-set programming," in *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*, L. P. Kaelbling and A. Saffiotti, Eds. Professional Book Center, 2005, pp. 90–96. [Online]. Available: http://www.ijcai.org/papers/1353.pdf

[14] J. J. Li, "Qualitative spatial and temporal reasoning with answer set programming," in *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012*. IEEE Computer Society, 2012, pp. 603–609. [Online]. Available: http://dx.doi.org/10.1109/ICTAI.2012.87

[15] L. Giordano, A. Martelli, and D. T. Dupré, "Reasoning about actions with temporal answer sets," *Theory and Practice of Logic Programming*, vol. 13, no. 2, pp. 201–225, 2013. [Online]. Available: http://dx.doi.org/10.1017/S1471068411000639

[16] V. Lifschitz, "Answer set programming and plan generation," *Artificial Intelligence*, vol. 138, no. 1-2, pp. 39–54, 2002. [Online]. Available: http://dx.doi.org/10.1016/S0004-3702(02)00186-8

[17] C. Baral, M. Gelfond, and J. N. Rushton, "Probabilistic reasoning with answer sets," *Theory and Practice of Logic Programming*, vol. 9, no. 1, pp. 57–144, 2009.

[18] R. A. Kowalski and F. Sadri, "From logic programming towards multi-agent systems," *Annals of Mathematics and Artificial Intelligence*, vol. 25, no. 3-4, pp. 391–419, 1999. [Online]. Available: http://dx.doi.org/10.1023/A:1018934223383

[19] M. Fink, S. Germano, G. Ianni, C. Redl, and P. Schüller, "Acthex: Implementing HEX programs with action atoms," in *Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, P. Cabalar and T. C. Son, Eds., vol. 8148. Springer, 2013, pp. 317–322. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40564-8_31

[20] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca, and T. Schaub, "ASP-Core-2 Input Language Format," March 2013. [Online]. Available: https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.01c.pdf

[21] A. Belov, D. Diepold, M. Heule, and M. Järvisalo, Eds., *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B. University of Helsinki, 2014, vol. B-2014-2, iSBN 978-951-51-0043-6.

[22] L. M. de Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[23] T. Frühwirth, *Constraint Handling Rules*. Cambridge University Press, Aug. 2009.

[24] International Conference on Planning and Scheduling, "PDDL Resources page," http://ipc.informatik.uni-freiburg.de/PddlResources/.

[25] J. Tiihonen, T. Soininen, I. Niemelä, and R. Sulonen, "A practical tool for mass-customising configurable products," in *Proceedings of the 14th International Conference on Engineering Design (ICED'03)*, Stockholm, August 2003, pp. 1290–1299.

[26] F. Calimeri and F. Ricca, "On the Application of the Answer Set Programming System DLV in Industry: a Report from the Field," *ALP Newsletter, Association of Logic Programming*, vol. 3, pp. 1–12, March 2012.

[27] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry, "An A prolog decision support system for the space shuttle," in *Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Intl. ASP'01 Workshop, Stanford, March 26-28, 2001*, A. Provetti and T. C. Son, Eds., 2001. [Online]. Available: http://www.cs.nmsu.edu/~tson/ASP2001/10.ps

[28] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca, and T. Schaub, "ASP-Core-2: 4th ASP Competition Official Input Language Format," 2013, https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.01c.pdf.

[29] F. Calimeri, G. Ianni, and F. Ricca, "The third open answer set programming competition," *Theory and Practice of Logic Programming*, vol. 14, no. 1, pp. 117–135, 2014. [Online]. Available: http://dx.doi.org/10.1017/S1471068412000105

[30] W. Faber, N. Leone, and G. Pfeifer, "Semantics and complexity of recursive aggregates in answer set programming," *Artificial Intelligence*, vol. 175, no. 1, pp. 278–298, January 2011.

[31] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, "The DLV System for Knowledge Representation and Reasoning," *ACM Transactions on Computational Logic*, vol. 7, no. 3, pp. 499–562, Jul. 2006.

[32] F. Buccafurri, N. Leone, and P. Rullo, "Enhancing Disjunctive Datalog by Constraints," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 5, pp. 845–860, 2000.

[33] L. Padovani and A. Provetti, "Qsmodels: ASP planning in interactive gaming environment," in *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings*, 2004, pp. 689–692.

[34] N. Leone and F. Ricca, "Answer set programming: A tour from the basics to advanced development tools and industrial applications," in *Reasoning Web. Web Logic Rules - 11th International Summer School 2015, Berlin, Germany, July 31 - August 4, 2015, Tutorial Lectures*, ser. Lecture Notes in Computer Science, W. Faber and A. Paschke, Eds., vol. 9203. Springer, 2015, pp. 308–326. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-21768-0_10

List of Figures