From Vision to Execution: Enabling Knowledge Representation and Reasoning in Hybrid Intelligent Robots Playing Mobile Games

Denise Angilica , Mario Avolio , Giovanni Beraldi
Giovambattista Ianni , Francesco Pacenza
University of Calabria
denise.angilica@unical.it, marioavolio@protonmail.com, gvnberaldi@gmail.com,
giovambattista.ianni@unical.it, francesco.pacenza@unical.it

Abstract

Automating acts on touch surfaces opens a range of possibilities for researching and experimenting with hybrid AI approaches. In this paper, we propose a delta robot capable of playing match-3 games and ball-sorting puzzles by acting on mobile phones. The robot recognizes objects of different colors and shapes through a vision module, is capable of making strategic decisions based on declarative models of the game's rules and of the game playing strategy, and features an effector that executes moves on physical devices.

Our solution integrates multiple AI methods, including vision processing and answer set programming. Helpful and reusable infrastructure is provided: the vision task is facilitated, while robot motion control is inherently simplified by the usage of a delta robot layout. We illustrate the components of our robotic application and how they were integrated. Then, we briefly showcase how recognition and general knowledge can be modeled and implemented, by overviewing the implementation of representative games.

We argue that our application provides potential for KR and robotics to be combined in creative ways, and offers itself as a general controlled environment where to experiment with forms of hybrid reasoning, while relieving from implementation details.

1 Introduction

A researcher in knowledge representation and reasoning (KRR in the following) that wants to approach robotics has to face many entry barriers: it is difficult to pinpoint the right hardware; the hardware itself can be out of the reach of available funding for a small laboratory, unless recurring to limited, educational robots; one has to address many implementation details requiring the presence of persons skilled in electronics and/or robotics in the research group; and, last but not least, one has to attend the tedious job of mapping quantitative and qualitative sensor and actuator data streams to the world of symbolic reasoning. On the other hand, symbolic reasoning almost entirely deals on high-level, qualitative information, and adapts itself with difficulty to, possibly continuous, numerical quantities.

We particularly focus in this paper on Answer Set Programming (ASP in the following), the known declarative paradigm with a tradition in modeling planning problems, robotics, computational biology as well as many other industrial applications (Erdem, Gelfond, and Leone 2016).

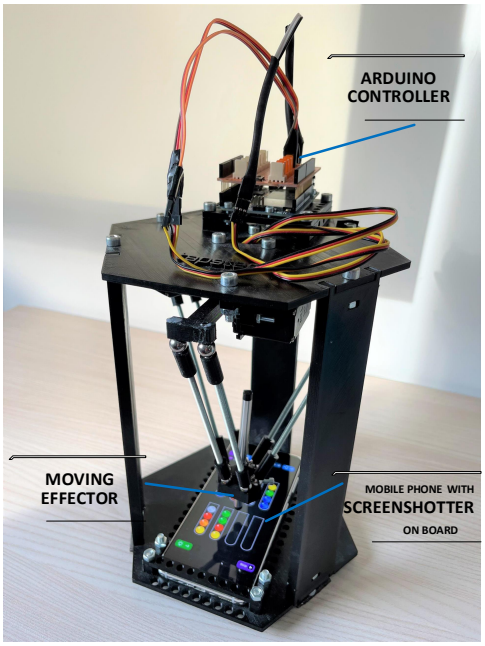


Figure 1: Main parts of the BrainyBot.

ASP shows a notable potential in robotics, and indeed its capabilities in terms of declarative problem solving, diagnostic and planning, have been helpful in many remarkable robotic applications, including coordination, path finding and planning for multi-robots, arrangement and assembly of object (Erdem and Patoglu 2018).

Applications of ASP in robotics are not exclusively of deductive nature, but can be also based on inductive techniques (Meli, Sridharan, and Fiorini 2021), while a non-exhaustive list of the fields of interest includes cognitive factories (Saribatur, Patoglu, and Erdem 2019), service robots (Chen, Yang, and Chen 2016) and autonomous surgical task planning (Meli, Sridharan, and Fiorini 2021).

In this paper we join the effort of leveraging ASP in robotics and enlarging the corresponding community, with the proposal of a controlled environment for experimenting and researching with mobile games.

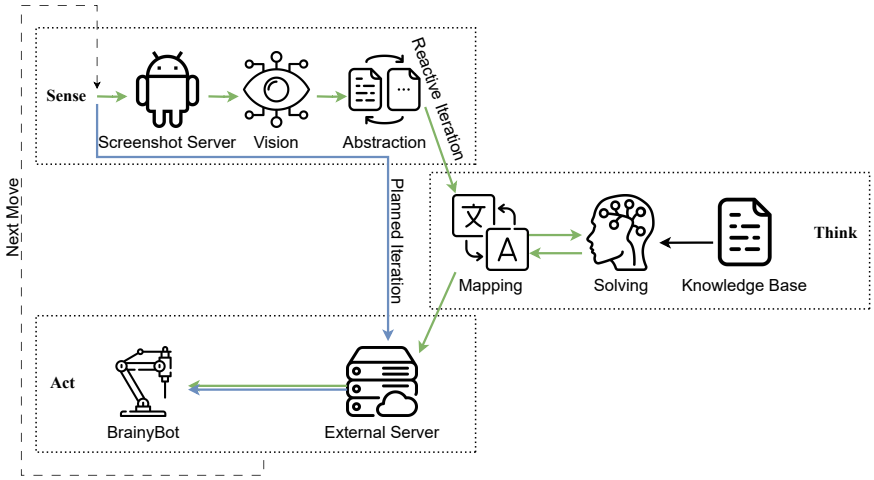


Figure 2: Main components of the BrainyBot.

The proposed BrainyBot bridges games to robotics, and eases research on the KRR aspects of these. Designers of BrainyBot appliances are relieved from many aspects of the sensor and actuator implementation and can focus on higher-level aspects of object recognition, discretization, abstraction and reasoning. Our main contributions are:

- we introduce a robot which acts on mobile phones to play match-3 games and ball-sorting puzzles;
- the robot integrates multiple AI methods, including vision processing and ASP, to automate the robot's actions on touchscreens;
- an ample part of the robot hardware and software is general enough to provide helpful and reusable infrastructure; this way a knowledge designer can better focus on reasoning aspects, since most of the low-level object recognition tasks are facilitated, while motion control is simplified by the adoption of a delta robot layout;
- the robot build is purposely inexpensive and both hardware and software are open-sourced: we hope this can contribute to expand the research community interested in reasoning and robotics, or generally interested in an experimental application bed for their research on KRR;
- we illustrate how recognition and general knowledge can be modeled in the framework of our robot. Also we report on how vision information can be translated to qualitative knowledge, and then how one can pass from qualitative reasoning to the execution of actions.

The paper is structured as follows: in Section 2, we describe the robot's components, its run-time workflow and how it interacts with mobile phone games; then in Section 3, we outline the types of games that the robot supports and the background knowledge required to play these games effectively; in Section 4, we discuss the design workflow of a new game AI on top of BrainyBot; in Section 5, we explain the vision and abstraction techniques used by the robot to recognize and interact with objects on mobile phones; in Section 6, we exemplify the declarative modelling techniques that can be used in a BrainyBot appliance to solve games; in Section 7, we discuss related work and then we outline future directions for research and conclusions in Section 8.

2 The Run-Time Workflow of BrainyBot

BrainyBot is built according to the design of the open source TapsterBot[1] project, which is also commercially distributed by Tapster Robotics. The robot is mostly 3D-printed and follows the known design of a delta-robot. Since their introduction (Vischer and Clavel 1998) delta-robots gained wide popularity for their relatively simple kinematics and ease of realization. Many variants of delta-robot exist (Clark et al. 2022; Kansal and Mukherjee 2022; Temel et al. 2018), and applications range from robotic surgery (Moustris and Tzafestas 2022) to agricultural (Yang et al. 2021). A delta-robot has a unique end-effector controlled in parallel by three arms: each arm is constituted by two rods forming a parallelogram. The arms can be moved by changing the angle of three respective servo motors. The design of the robot is such that the orientation of the effector is kept virtually constant, while one can drive its 3D position.

The main hardware parts of a BrainyBot (Figure 1) are a mobile device PH, the effector E controlled using an Arduino board, and a computer C (not shown). A touch stylus is placed at the center of E. Figure 2 shows the operating workflow of an instance of a BrainyBot, which is conceived according to a mix of the classic sense-think-act loop and its hybrid-deliberative variant (Murphy 2000). Software components are placed respectively on PH or on C. In turn, C controls all the parts of the system. A game G of choice runs on PH. From the perspective of a BrainyBot instance, one can regard the touch display of PH as the external environment from which information is sensed, and to which the actions of BrainyBot are directed. BrainyBot cyclically processes information taken from PH's display, then decides and executes arm moves on the touch display itself. More

[1] https://github.com/tapsterbot/tapsterbot

in detail, in each iteration, the Sense-Think-Act workflow is executed as described next.

Sense. Along with the game G, a `ScreenshotServer` runs on PH and is prompted to produce an image *IM* of what currently appears on PH's display. *IM* is processed by a vision module, which recognizes relevant objects together with their raw position and other attributes like color and shape. Before sending this information to the next step, an `Abstraction` module transforms quantitative information, (e.g., pixel coordinates of objects) into an abstracted description of the current game board. This operation is performed by exploiting some game background information, like assuming that the game level contains objects forming a rectangular grid, almost default in match-3 games, or assuming that objects are laid out in stacks, a typical setting of ball sort games. Background knowledge is also helpful in filtering out false positive images, such as duplicates, or ghost detections appearing in displaced positions.

Think. The task of this step depend on whether the current iteration is *planned* or *reactive*: the two different workflows are depicted in Figure 2 by the blue and green paths, respectively. In reactive iterations the decision making step is performed inline whereas, in planned iterations the robot executes moves that were planned ahead. In this latter type of iteration, the think and vision steps are skipped. In a reactive iteration, the game board description is transformed in logical assertions. Logical assertions are then fed to the `Solving` module together with a declarative representation of the game rules and of the game strategy of choice. The `Solving` module is made of an answer set solver S interfaced using the EmbASP framework (Calimeri et al. 2019), which plays the role of `Mapping` module. S is expected to produce at least one optimal *answer set* A. A might encode the next move to be executed, or a sequence of moves to be executed in later planned iterations.

Act. Eventually, a move appearing in A is converted back into a specific gesture to be executed on the touch display. A gesture can be either a tap or a swipe, and requires to specify proper quantitative pixel coordinates. Specifically, taps require to specify a couple of screen coordinates (x, y) where to perform a tap, whereas swipes require a 4-tuple (x_0, y_0, x_1, y_1) describing a line to be swiped while touching the screen. The inverse and forward kinematic of a delta-robot has been widely studied since its proposal (Vischer and Clavel 1998). Given a desired target point $T = (x, y)$ to touch on the mobile surface, T is first converted to a tridimensional value $T' = (x_0, y_0, z_0)$ using a function $C(T, d) = T'$ where d are calibration data. It is then possible to make the robot effector reach T' by using an inverse kinematics function $F(x_0, y_0, z_0) = (\theta_1, \theta_2, \theta_3)$ where $(\theta_1, \theta_2, \theta_3)$ are the obtained target angles for the servo motors (Figure 3).

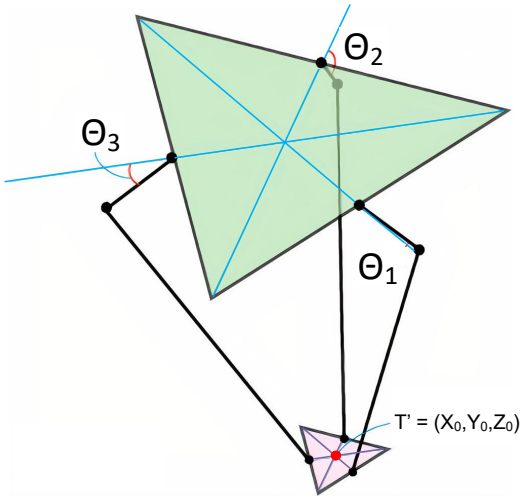


Figure 3: Abstract layout of a delta-robot.

3 Types of Games and Background Knowledge

Before describing how one can design an AI working for a BrainyBot, we briefly overview the features of the games we focused on in our research. Our BrainyBot prototype can currently play two types of games: *Match-3* and *Ball Sort* games. Both game types are conceived for individual players which are challenged to solve a game level, whose initial and following states are shown on a screen. The gameplay is turn-based, and there is, in general, no time limit between a move and another. The two categories of games differ however in the board layout, in the level of observability and in the game strategy. On the one hand, Ball Sort puzzles have a peculiar board layout, formed of stacks of objects; they are usually complete information games with a shallow search tree. On the other hand, Match-3 games are typically not fully observable and feature some non-deterministic elements in them, thus being an interesting ground for research in the game-play strategy.

We briefly outline both game types next, with particular focus on some standard assumptions that can help in modelling *a)* the game rules and the decision-making process, and *b)* the recognition of the game board and the subsequent abstraction steps.

Match-3 games. The family of Match-3 games takes its name from its basic move, consisting in swapping objects laid out in a planar grid, and aiming to align at least three objects of the same kind. The most representative game of this family is Candy Crush Saga[2] (CCS in the following).

Game rules. In Match-3 games, legal moves require to align at least three identical objects by swapping one source object s with a vertically/horizontally adjacent object t. After a move, all the objects involved in alignments of three or more of the same kind/color disappear, causing the objects above to fall down into the now empty space, and creating difficulties in predicting chain reactions.

Images and Board layout. Simpler objects in Match-3 games use a fixed shape and color: for instance, in CCS there are orange oval candies, green square candies, etc. Simple objects can have special attributes, which change their appearance, e.g., there can be candies blocked in a liquorice

[2]https://www.king.com/it/game/candycrush

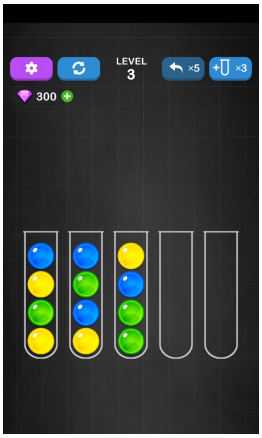


Figure 4: A Ball Sort game screenshot.

cage. All the objects are arranged in a planar grid g. Although g is not necessarily square, all the objects in the game are laid out according to discrete rows and columns.

Ball Sort puzzles. Ball Sort Puzzles constitute a family of smartphone video games available in several variants from different software houses.

Game rules. In general, a level of a ball sort puzzle consists in $n+m$ tubes, n of which are filled each with p stacked balls which can have one out of n colors, while the remaining m tubes are empty (as shown in Figure 4). A maximum of p balls can be in a certain tube at any given moment. A game level is solved when one has n full tubes, each with balls of the same color. The core rule of the game prescribes that only a ball on top of its source tube s can be moved to another target tube t, provided that t contains at most $p-1$ balls before the move and that the ball on top of t has the same color of the one on top of s.

Images and Board layout. Ball Sort puzzles feature two type of objects: balls and tubes. Balls can differ in their color, whereas tubes have a fixed shape, although their appearance might vary depending on whether they contain balls or not. The structure of the game board is nested: balls are stacked within tubes, while tubes are arranged in discrete rows and columns.

4 BrainyBot Design Workflow

Let us assume that a researcher R aims to use a BrainyBot for research purposes, maybe customizing its workflow for a new mobile game/application G. One can proceed by modifying/reusing the modules shown in Figure 2 as follows.

Camera taking and screenshots. BrainyBot does not purposely feature a physical camera. Taking screenshots is simplified as it is sufficient to install the Screenshot Server application on the mobile phone at hand.

Vision. The Vision module includes the builtin possibility of recognizing objects of fixed appearance, typical of Match-3 games, or balls of different colors, typical of Ball Sort puzzles. When implementing the AI for G, R can add new template images corresponding to new arbitrarily shaped objects. Objects in the shape of balls can be instead automatically detected whenever they appear in a screenshot, independently from their color and size.

Abstraction. R can use predefined modules able to reconstruct discrete planar grids of objects, typical of Match-3 games, or groups of stacks of objects, typical of Ball Sort Puzzles. With relatively small effort, the Abstraction module can be customized for accommodating differences in the layout of the game at hand, possibly implementing arbitrary board layouts.

Mapping. The implementation of BrainyBot provides predefined data structures which are automatically mapped to predefined predicates. These allow to model object grids and content thereof, balls and tubes and their layout, and associate them to their mapped logic assertions. For instance, the Python class `Edge` is associated to the homonymous predicate used for modelling grid adjacency graphs. If necessary, the EmbASP subsystem allows to easily map new custom logic predicates to and from Python objects meant to describe other features of a game level. For instance, given a game based on a maze structure, one could be interested in modelling the `Wall` class: this could be automatically mapped to a logic assertion with a few lines of Python code.

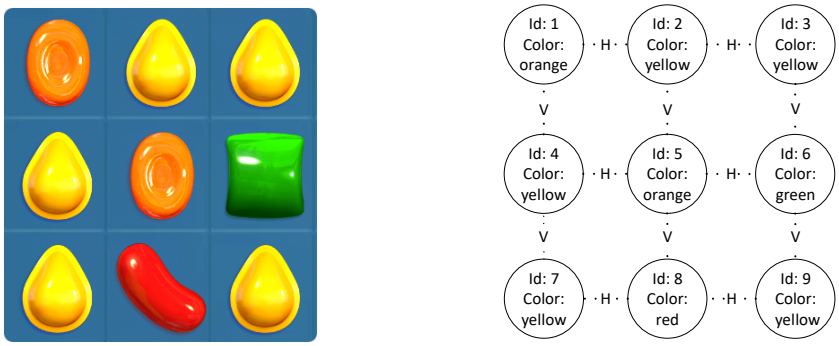
Solving. Game rules and decision making can be declaratively modelled by providing a declarative knowledge base KB written in ASP, which combined with a game board description B, and run using an ASP solver, would produce appropriate game moves. This is the most flexible part of the BrainyBot infrastructure, as R can experiment with different knowledge bases, in order to accommodate diverse game rules and diverse game playing strategies.

Acting. R is relieved from most of low-level details of the acting step, except for the customary calibration routine. Calibration requires to redefine the function $C(.,d)$ with updated values for d, meant to compensate various factors that can cause the position and orientation of the three arms to drift. This process must be repeated periodically and is partially manual.

5 Vision and Abstraction Techniques

The Vision subsystem uses known computer vision methods, available in the OpenCV library (Bradski 2000) to recognize type and position of the game elements appearing in-game screenshots. Each of these techniques might be specific to the type of game at hand: a useful common technique is Template Matching (TM, in the following) which allows the recognition of fixed objects. In TM, one tries to locate one or more instances of a template image *im* within a larger-sized image *IM*. To achieve this goal, *im* is iteratively translated onto the larger image, thus performing a 2D convolution. An output matrix *m* of size $(W-w+1, H-h+1)$ is returned, where (W,H) represents the size of the larger image, while (w,h) represents the size of *im*. Each element of *m* denotes how much, at any valid coordinate (x,y) of *IM*, *im* coincides with the corresponding subregion of *IM*.

Note that *m* typically reports several matchings very close to each other. We use the normalized cross-correlation coefficient to score matchings, and we consider only matches above a given threshold. A match m is identified by its center (x_m, y_m). As there can be several partial matchings of



(a) (b)

Figure 5: Game graph representation.

the same image, with just a slight difference in their coordinates, we use a deduplication strategy in which a new match m' is assumed to be a duplicate of m, and thus discarded, whenever m' coordinates $(x_{m'}, y_{m'})$ fall in the bounding box of m. Considering that game objects very often have a static shape in the type of turn-based games we are targetting, we opted for TM as it works very well on fixed images and does not require time-consuming training and fine-tuning.

Recognition of Match-3 objects and grids. The vision module in charge of reconstructing levels of a Match-3 game makes use only of TM to recognize objects in a screenshot. We implemented our recognition strategy in the specific case of CCS. All the candy object template images are stored and labeled with a type identifier that denotes the corresponding candy throughout the game. The threshold of choice for comparing normalized cross-correlation values (0.85) has been obtained by looking at the highest value with zero false negatives. In the next abstraction layer, background knowledge is used very effectively in discarding the possibly remaining false positives (e.g. out-of-grid artifacts), thus virtually achieving a precision and recall of 100%.

Each candidate object o has its own pixel coordinate (x_o, y_o) and a type t. The Vision module forms a collection O of candidate objects. Then Abstraction works by arranging elements of O in a grid g, modeled as an undirected graph connecting adjacent objects (Figure 5), either in a vertical or horizontal relationship. We assume each template image (*sprite* in the following) is of the same size. We form the nodes and edges of g by taking the left-uppermost object $o \in O$ and sweeping the pixel space in fixed vertical and horizontal steps of the size of a sprite. A horizontal/vertical link between two nodes n_1 and n_2 is added only if the distance between their centers is around the size of a sprite. The coordinates of unique sprite occurrences are replaced with a unique identifier, while edges are labeled to take into account the horizontal/vertical adjacency relationship.

Recognition of balls, tubes and puzzles layouts. Since Ball Sort puzzles suit the approach of planning in advance all the solution moves, we take a screenshot for the initial game board only. Among the current implementations of this puzzle, we used *Ball Sort Puzzle - Color Game*[3] as test case. The subsystem that can reconstruct Ball Sort game levels combines a number of vision techniques.

Tubes. We use sprites of empty tubes as templates to be identified within the game snapshots. These have been directly acquired from game screenshots. We set the threshold for the normalized cross-correlation coefficient to 0.80, again choosing the highest value with no false negatives, which experimentally gave us 100% precision and recall of empty tubes appearing in a screenshot *IM*. Tubes of different sizes and height are detected using a set of different matching images.

Balls. Balls are game objects constituting a typical example of image that cannot be recognized by simple template matching. Their size, texture and color can be very different from one level to another, and it is not feasible to maintain stored template images for any possible visual appearance of this kind of object. Ball detection requires two separate vision tasks: coordinate detection and color detection.

For coordinate detection, we applied a Canny Edge Detector (Ruff 1987) to *IM*, to obtain contours of game objects, then the Hough Circle Transform (Yuen et al. 1989), a specialization of the Hough Transform that is used in the field of digital image processing, to identify circles present in an image. All pixels in the screenshot are ranked according to their likelihood of being at the center of a circle. It is important to note that some circle centers may be mistakenly detected in the presence of edges that define shapes similar to a circle, such as the lower part of tubes that describe a semicircle. This issue is solved by the `Abstraction` module, which is able to filter the detection results.

In order to detect ball colors, one has to take into account small differences in hue and texture and some lack of precision in finding the precise center of a given ball. This issue is circumvented by applying a Gaussian blur filter to *IM*. This filter smoothens noise and decreases the level of detail, thus making decorative patterns lesser relevant in color detection. The ball color is then detected by looking at the blurred image instead of the original one.

Layouts and abstraction. The Abstraction module filters

[3] https://bit.ly/ball_sort_puzzle_android2

out false positives and determines qualitative values for balls and tubes' positions, and for ball colors.

Concerning colors, the Abstraction module works assuming a dynamic number of ball colors are present in the game. A collection of color labels C is constructed as follows: when a new RGB value (r, g, b) is detected for a candidate center pixel of a ball, we assume to be in the presence of a new color if the minimum Euclidean distance in the RGB space from colors in C exceeds a given threshold. Otherwise, we map (r, g, b) to the element $c \in C$ which has the closest Euclidean distance in the RGB space (see Figure 7).

The quantitative information associated with the pixel position of balls is abstracted using an algorithm that iterates over the raw data received as input, i.e., the pixel coordinates of the individual balls, and instantiates for each coordinate pair a Ball object, to which it is assigned the correct color label $c \in C$.

Each tube t is modeled using a unique label and a list L_t of contained balls. Empty tubes which were separately identified using TM are instantiated with an empty list of balls, while tubes containing balls are implicitly identified using the position of detected balls.

In order to reconstruct the content of a tube t, we assume it respects the following criteria: i) contained balls are vertically aligned, i.e., their x coordinate may be not perfectly aligned but belong to a fixed pixel range; ii) the difference between the y coordinate of each ball of t must be lower than a certain threshold τ_1 and not higher than a second threshold τ_2; this permits to tell apart ghost detections and balls coming from different tubes which are aligned vertically (Figure 8). Let B be the set of detected balls. The content of a new tube t is reconstructed by moving a ball b from B to t. Then we move from B to t all the balls $b' \in B$ that fulfill the above distance criteria from elements of t. New tubes are created until B is empty, obtaining a collection T of tubes.

The content of each tube represents an ordered stack where the original pixel position of balls is abstracted away. Since tubes contain at least 4 balls, we assume all $t \in T$ with less than 4 balls are ghost detections, and we remove them from T.

Figure 6: A Candy Crush game screenshot with yellow candies highlighted by the Vision module.

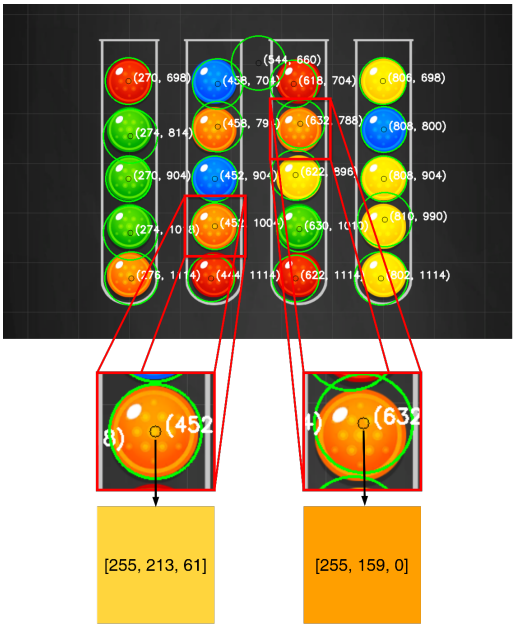


Figure 7: Color matching in the Ball Sort game.

6 Declarative Knowledge Modelling

In this section we overview the knowledge modeling process happening when designing the Think stage: in particular we showcase how parts of the game logic and the decision process can be declaratively described using ASP.

We herein recall briefly the main features of ASP (Gelfond and Lifschitz 1988; Gelfond and Lifschitz 1991) the known declarative logic formalism developed in the field of logic programming and non-monotonic reasoning. The basic constructs of ASP are rules with form *Head* :- *Body*, where *Body* is a logic conjunction in which negation may appear, and *Head* can be either an atomic formula or a logic disjunction. ASP knowledge bases are composed of set of rules. *Choice rules* allow to define spaces of possible values for logical assertions, while *hard* and *soft constraints*, allow respectively to define disallowed or undesirable scenarios. Soft constraints can be prioritized, thus enabling the possibility of combining multiple optimization criteria on several tiers. One can model problems in ASP by expressing knowledge into three distinct parts, namely *guess* (mostly composed of choice rules and disjunctive statements), *check* (mostly composed of hard constraints) and *optimize* (mostly composed of soft constraints).

A set of input values F (called *facts*), describing the current state of the world, are fed together with an ASP knowledge base *KB* to a *solver*. Solvers in turn produce one or more outputs $A(KB \cup F)$ called *Answer Set(s)*. An answer set is a set of logical assertions, which might represent shifts to be scheduled, protein sequences, diagnoses, and so on. In our setting an answer set encodes a set of one or more moves to be executed on the game board at hand. If a program has no answer sets, the corresponding input problem has no solutions. We refer the reader to (Calimeri et al. 2020) for an analytic description of the full syntax and semantics of the

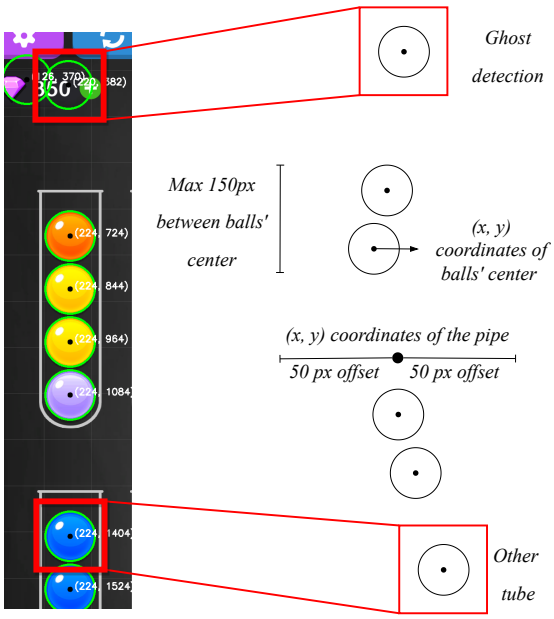


Figure 8: Checks on tubes detection.

ASP-Core-2 standard language. A number of solvers and systems are available among which Clingo (Gebser et al. 2011; Gebser et al. 2019) and DLV2 (Calimeri et al. 2016; Calimeri et al. 2022) are actively developed.

Next, we overview how one could model games, by focusing on some of the salient parts of knowledge bases meant to declaratively specify the rules and the game playing strategy for the Match-3 and Ball Sort Puzzle families. Note that the focus of this paper is not on the quality of the presented declarative AIs: these are assumed to be modular, and expected to be provided by interested KRR designers.

Match-3 games. We designed an illustrative ASP-based Think module focused on CCS. CCS follows the basic rules of Match-3 games, i.e., one can swap adjacent objects provided they make an alignment of three or more objects of the same kind (i.e., they produce a *3-match*). Matches imply cancellation and fall of objects on the game board. Default objects are candies of several color and shapes, but special objects are possible. Swapping special objects, like e.g., the so-called *color bomb*, has specific special effects. For the sake of simplicity we will not show the modelling of the behavior of special objects. The current game board is modeled via a non-oriented graph, in which nodes represent game objects, while edges model adjacency relations among them. In the case of CCS, game objects correspond to candies of several shape and color (see Figure 6). The game graph is mapped to input facts over the predicates `node` and `edge`. A fact of the form `node(id,t)` is used to model that the game object having as unique identifier `id` is a node of type `t`; in CCS, the type is the shape and color of the candy. A fact of the form `edge(id1,id2,A)` models that there is an alignment between the directly adjacent nodes `id1` and `id2`, where A can be either v or h, meaning respectively a vertical or horizontal alignment, as shown in Figure 5b. The following ASP rules can be used for defining the basic game rules and the optimal object swap.

```
r1 : {swap(ID1,ID2): edge(ID1,ID2)}=1.
r2 : swap(ID2,ID1) :- swap(ID1,ID2).
r3 : newEdge(ID1,ID3,A) :-
        swap(ID1,ID2), edge(ID2,ID3,A),
        ID1!=ID3, node(ID1,T), node(ID3,T).
r4 : deletedEdge(ID2,ID3) :- edge(ID2,ID3,A),
        swap(ID1,ID2).
r5 : match(ID1,ID3,A) :-
        newEdge(ID1,ID2,A),
        newEdge(ID2,ID3,A), ID1!=ID3.
r6 : match(ID1,ID3,A) :-
        newEdge(ID1,ID2,A),
        edge(ID2,ID3,A), not deletedEdge(ID2,ID3),
        node(ID2,T), node(ID3,T), ID1!=ID3.
r7 : match(ID1,ID3,A) :-
        match(ID1,ID2,A),
        edge(ID2,ID3,A), not deletedEdge(ID2,ID3),
        node(ID2,T), node(ID3,T), ID1!=ID3.
```

Rule r_1 defines a search space among possible swaps between two game objects. Rule r_2 makes a swap as a symmetric operation (i.e., swapping nodes a and b implies swapping b and a). Since swapping nodes make edges change, rules r_3 and r_4 are respectively used to determine new edges and keep track of no longer existing connections. Rules r_5 and r_6 determine simple matches, i.e., configurations of three aligned game objects of the same type, whereas rule r_7 recursively defines longer strides of matches.

Constraint r_8 ensures that only swaps provoking at least a 3-match are allowed:

```
r8 : :- #count{ID1,ID3,A: match(ID1,ID3,A)}=0.
```

Among all admissible swaps, one can declaratively specify the preferred one in terms of some custom criterion. For instance, soft constraint r_{12} below attributes a cost to swaps:

```
r9 : involvedNode(ID) :- match(ID,_,_).
r10 : involvedNode(ID) :- match(_,ID,_).
r11 : survivingNode(ID) :- node(ID,_),
        not involvedNode(ID).
r12 : :~ survivingNode(ID). [1@1,ID]
```

This cost is proportional to candies remaining in the scene after the swap, thus defining that the more desirable solutions are those with the highest number of destroyed game objects. The rules and game strategy for Match-3 games can be customized at will by adding additional knowledge and other soft/hard constraints.

Ball Sort Puzzles. We show next how one can model game rules and the decision making process for ball-sort puzzles. We devised a knowledge base that can find an execution plan at once or be broken into layered pieces deciding shorter sequences of steps. At each step a single move is made and the ultimate goal is to minimize the total number of moves. The initial state of the game is modeled via some ASP facts over the predicates `color`, `ball`, `tube`, `tubeSize`, and `on`. Facts of the form `color(c)` describe the possible colors. A fact of the form `ball(b,c)` represents that the ball `b` has color `c`. Facts of the form `tube(t)`

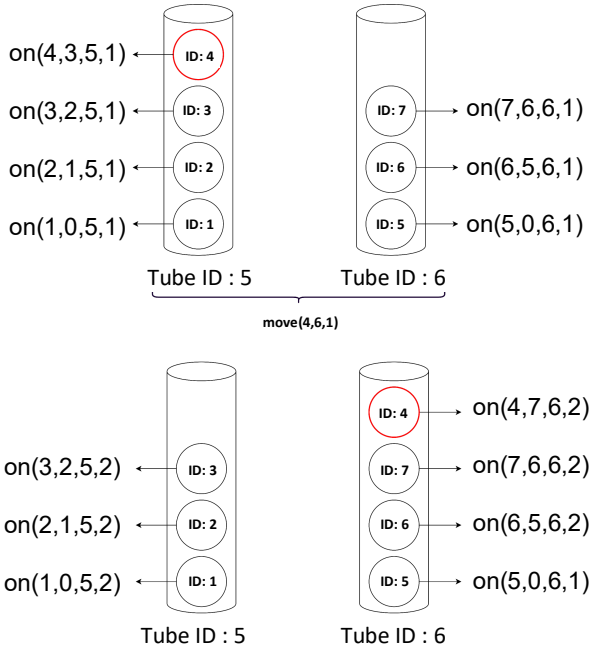


Figure 9: Effect of a legal move in a Ball Sort game.

describe the tubes identified in the game scene, whereas the fact `tubeSize(s)` expresses the global size of tubes. Finally, a fact of the form `on(b1,b2,t,s)` encodes that the ball `b1` is above the ball `b2` and both belong to the tube `t` at step `s`. The *guess* portion of the problem model, introduces per each step s, a possible move until the level is completed (rule r_1) and determines the game state at step $s+1$ (rules r_2–r_4). These latter rules use some predicates defined by simple auxiliary rules, of which we omit the description as their meaning is intuitively given by the names of predicates; more in detail, r_2 states that, in the next step, a ball b_1 is on top of a ball b_2 if b_1 is moved in the tube t and b_2 was on top of t; r_3 says that when a ball b_1 is moved into an empty tube, it is the lowest one (the special placeholder 0 encodes that there is no other ball under b_1); r_4 is used to determine the state of all other balls not involved in a movement in the current step (Figure 9). r_1–r_4 are as follows:

```
r1 :  {move(B,T,S): onTop(B,T1,S), tube(T), T!=T1}=1 :-
          step(S), not levelComplete(S).
r2 :  on(B1,B2,T,S+1) :-
          step(S), move(B1,T,S), onTop(B2,T,S).
r3 :  on(B1,0,T,S+1) :-
          step(S), move(B1,T,S), emptyTube(T,S).
r4 :  on(B1,B2,T,S+1) :-
          step(S), on(B1,B2,T,S), not ballMoved(B1,S).
```

The *check* part consists of strong constraints that discard not allowed plans. In particular, at each step s, a move is not admissible when either: i) b is moved in a full tube (rule r_5) or, ii) b has been moved in the previous step $s-1$ which would result in a useless move (rule r_6) or, iii) b is moved on top of another ball b' with a different color (rule r_7):

```
r5 :  :- step(S), move(B,T, S), size(T,S,N), tubeSize(N).
r6 :  :- step(S), move(B,_,S), move(B,_,S-1).
r7 :  :- step(S), move(B, T, S), onTop(B1,T,S), C1!=C2,
         ball(B,C1), ball(B1,C2).
```

The *optimize* part of the program defines the game strategy by means of soft constraints whose priority depends on the associated tier level (i.e., the higher the level, the greater the priority). It is worth to note that this part is the one more strictly dependent to the specific Ball Sort game variant, but it lends itself to be repurposed to other games of the same category. We list below some possible soft constraints, ordered in descending priority. We say that a tube is mono-color if it contains only balls of the same color. In rule r_8, we introduce a cost if we don't move a ball of color c in the mono-color tube containing the largest number of balls of the same color c. We prefer to move a ball in an empty tube, if available, in order to facilitate the construction of a mono-colored tube (rule r_9) and among all colors, we prefer to move a ball of a color c, which appears on top of the largest number of tubes (rule r_{10}). Via rule r_{11}, we discourage to move a ball of color c which is already in the tube containing the largest number of c-colored balls. Rule r_{12} aims at freeing balls of a color c that are not on top of a tube when there is a tube having only c-colored balls. Rule r_{13} suggests to move a ball having another one of the same color below it. Finally, rule r_{14} encourages to move a ball in a tube t that in the next step would keep t mono-colored. r_8–r_{14} are as follows:

```
r8 :  :~ step(S), not move(B,T1,S), not inTube(B,T1,S),
         biggestMonoColorTube(T1,C,S), ball(B,C),
         onTop(B,_,S). [1@7, B,S]
r9 :  :~ step(S), move(B,T1,S), not monoColorTube(T1,S),
         notEmptyTube(T1,S), emptyTube(T2,S),
         T1!=T2. [1@6,B,S]
r10 : :~ step(S), move(B,_,S), ball(B,C),
         ballOnTopOfColor(C,N,S), C!=C1, N<M,
         ballOnTopOfColor(C1,M,S). [1@5,B,S]
r11 : :~ step(S), inTube(B,T1,S), ball(B,C),
         biggestMonoColorTube(T1,C,S),
         not biggestMonoColorTube(T2,C,S),
         move(B,T2,S). [1@4,B,S]
r12 : :~ step(S), move(B,_,S), onTop(B1,T,S),
         not monoColorTube(T,S), on(B1, B2,T,S),
         ball(B2,C), monoColorTubeWithColor(_,C,S),
         B!=B1. [1@3,B,S]
r13 : :~ step(S), move(B,_,S), on(B,B1,T,S),
         not monoColorTube(T,S), ball(B,C),
         ball(B1,C1), C!=C1. [1@2,B,S]
r14 : :~ step(S), move(B,_,S), on(B,_,T,S),
         not monoColorTube(T,S+1). [1@1,B,S]
```

7 Related Work

Our work contributes to other lines of research in several respects, that can be categorized as *(1)* Research on Match-3 and Ball sort games; *(2)* Vision in games, both physical and on screen; *(3)* Other end-to-end approaches integrating ASP; *(4)* Modelling games in ASP.

Research on Match-3 and Ball sort games. Match-3 games stimulated interest in the general research community since it features elements of non-observability which makes the task of game playing nonobvious. Also prescribed game goals can be fairly complex, ranging from destroying given target objects to moving objects in given target locations in a limited number of moves. Match-3 have been proved to be NP-hard in (Gualà, Leucci, and Natale 2014), and indeed their combinatorial properties are nontrivial (Hamilton, Nguyen, and Roughan 2021). Given its wide popularity it is rather unsurprising that CCS has been subject of social research (Chen and Leung 2016; Amaro, Veloso, and Oliveira 2016). Ball Sort puzzles and their variants in which quantities of water are sorted, can be seen as a gamified variant of the classic Hanoi Tower puzzle. Ball and water sort puzzles have been proved to be NP-Hard (Ito et al. 2022); a usage of Ball Sort as a testbed in augmented reality can be found in (Maraj, Hurter, and Murphy 2020).

Vision in games. In the context of artificial game player programming, it is necessary to know the game scene in order to obtain a credible AI: many interesting commercial games do not expose an abstract representation of the game scene thus the vision task can not be neglected. In the Angry Birds competition (Renz et al. 2015) the organizers deal with this aspect and provide participants with a vision module producing and encoding the current game scene. Even when the main objective is not to obtain an artificial player, vision in video games can be helpful. In (Gielis et al. 2019) authors used the game FreeCell for the diagnosis and follow-up of cognitive impairments, using image recognition techniques for extracting digital bio-markers. When it comes to robots playing against humans on physical board games, a lot of studies have been conducted on detecting game situations on the basis of scenarios captured by cameras, with special focus on Chess (Wei et al. 2017; Wölflein and Arandjelovic 2021) and Go (Scher, Crabb, and Davis 2008; Song and Li 2018).

Other end-to-end approaches integrating ASP. Our work shares the same spirit of (Andres et al. 2015), in which facilities for connecting ASP to ROS-enabled (Quigley et al. 2009) robots are provided. The adaptation of the sense-think-act cycle follows the ideas of ThinkEngine (Angilica, Ianni, and Pacenza 2022), in which we proposed a similar variant of the evaluation loop; this allows to program non-player characters in video games by mixing plans and reactive actions. The necessity of acting on planar surfaces is not restricted to the case of touch surfaces. For instance the surgical Vinci® robot has been used in (Meli, Sridharan, and Fiorini 2021) by inducting ASP programs and then testing and training on a planar surface, which substitutes the patient open abdomen with an abstract description thereof. We plan to investigate the usage of a tablet device as a simplified environment for testing surgical robots.

Game modelling in ASP. ASP has a longstanding tradition in modelling single player, turn-based games. One can mention all the game-based problems appearing in the past ASP Competitions from its first through its seventh edition (Gebser et al. 2007; Gebser, Maratea, and Ricca 2020) or even work combining or using ASP in video games (Calimeri et al. 2016; Stanescu and Certický 2016; Angilica, Ianni, and Pacenza 2022). This research has a clear connection with general game playing languages and GDL in particular (Genesereth et al. 2005): indeed ASP has been attempted to be used as a general game description language in (Thielscher 2009; Smith et al. 2010).

We believe our work can stimulate this line of research as it opens many possibilities such as *(i)* experimenting with new games, *(ii)* collect new benchmark data taken from actual levels, *(iii)* investigate on practical abstraction techniques, *(iv)* explore what can be the impact of the presence of relatively complex gestures in the reasoning workflow, such as swipes, taps, long and double taps, etc., and, *(v)* last but not least, one should not disregard the emotional impact of "seeing things done by my robot" that can spark further interest in games and robotics research & learning.

8 Conclusions

In this paper, we have presented a robot appliance aiming to bridge the gap between knowledge representation, games and robotics. The proposed BrainyBot can serve as a generic ground where a researcher can focus on high-level aspects of object recognition, discretization, abstraction, and reasoning, while lower level implementation details are fairly simple to address, thus reducing the barriers to entry for researchers interested in this type of application.

Many components of our contribution are reusable, including standardized objects, grid reconstruction algorithms and (portions of) declarative models written in ASP. The inexpensive and open-source nature of our robot's hardware and software is intended to expand the community of researchers interested in reasoning and robotics, offering a practical application bed for research on knowledge representation and reasoning. Through this work, we have demonstrated how vision information can be translated into qualitative knowledge, and how qualitative knowledge can inform actual actions.

Future research directions include developing support for additional game categories, enlarge the portfolio of available standard objects and board reconstruction tools, automate handling of interstitial game screens (recognition and pushing of "next level" buttons, etc.), and enlarge the range of BrainyBot capabilities to mobile applications. Also, we plan to further develop the knowledge representation techniques used in conjunction with the robot in many directions. Among these we aim to use BrainyBot as an application and testing environment where to represent and deal with continuous numeric values. BrainyBot qualifies also as a good data producer and consumer for ongoing search on stream and incremental reasoning (Ianni, Pacenza, and Zangari 2020; Calimeri et al. 2021; Calimeri et al. 2022). In the actuator design compartment, the BrainyBot could be improved by eliminating the need for human involvement in the tedious calibration process and incorporating feedback mechanisms to more effectively handle plan failures and replanning. The BrainyBot building instructions, its control software and declarative knowledge bases are available at https://github.com/DeMaCS-UNICAL/BrainyBot.

Acknowledgments

We thank the reviewers of this paper, whose useful feedback helped to improve our work. This work was partially supported by: the PNRR MUR project PE0000013-FAIR, Spoke 9 - Green-aware AI – WP9.1; the project "Declarative Reasoning over Streams" (CUP H24I17000080001) – PRIN 2017 and by the LAIA laboratory (part of the SILA laboratory network at University of Calabria).

References

Amaro, A. C.; Veloso, A. I.; and Oliveira, L. 2016. Social games and different generations: A heuristic evaluation of Candy Crush Saga. In *TISHW*, 22:1–22:8. IEEE.

Andres, B.; Rajaratnam, D.; Sabuncu, O.; and Schaub, T. 2015. Integrating ASP into ROS for reasoning in robots. In *LPNMR*, volume 9345 of *LNCS*, 69–82. Springer.

Angilica, D.; Ianni, G.; and Pacenza, F. 2022. Declarative AI Design in Unity using Answer Set Programming. In *CoG*, 417–424. IEEE.

Bradski, G. 2000. The openCV library. *Dr. Dobb's j. softw. tools prof. program.* 25(11):120–123.

Smith, A.M.; Nelson, M. J.; Mateas, M. 2010. LUDOCORE: A logical game engine for modeling videogames IEEE CIG 91–9.

Genesereth, M.R.; Love, N.; Pell, B. 2005. General Game Playing: Overview of the AAAI Competition AI Mag. 26(2):62–72.

Calimeri, F.; Fuscà, D; Perri, S.; and Zangari, J. 2016. I-dlv: The New Intelligent Grounder of dlv. In *AI*IA*, volume 10037 of *LNCS*, 192–207. Springer.

Calimeri, F.; Fink, M.; Germano, S.; Humenberger, A.; Ianni, G.; Redl, C.; Stepanova, D.; Tucci, A.; and Wimmer, A. 2016. Angry-HEX: An artificial player for Angry Birds based on declarative knowledge bases. *IEEE Trans. Comput. Intell. AI Games* 8(2):128–139.

Calimeri, F.; Fuscà, D.; Germano, S.; Perri, S.; and Zangari, J. 2019. Fostering the use of declarative formalisms for real-world applications: The EmbASP framework. *New Gener. Comput.* 37(1):29–65.

Calimeri, F.; Faber, W.; Gebser, M.; Ianni, G.; Kaminski, R.; Krennwallner, T.; Leone, N.; Maratea, M.; Ricca, F.; and Schaub, T. 2020. ASP-Core-2 input language format. *Theory Pract. Log. Program.* 20(2):294–309.

Calimeri, F.; Ianni, G.; Pacenza, F.: Perri, S.; and Zangari, J. 2019. Incremental Answer Set Programming with Overgrounding. *Theory Pract. Log. Program.* 19(5-6):957–973.

Calimeri, F.; Manna, M.; Mastria, E.; Morelli, M. C.; Perri, S.; and Zangari, J. 2021. I-DLV-sr: A Stream Reasoning System based on I-DLV. *Theory Pract. Log. Program.* 21(5):610–628.

Calimeri, F.; Ianni, G.; Pacenza, F.: Perri, S.; and Zangari, J. 2022. ASP-based Multi-shot Reasoning via DLV2 with Incremental Grounding. In *PPDP*, 2:1–2:9. ACM.

Chen, C., and Leung, L. 2016. Are you addicted to Candy Crush Saga? An exploratory study linking psychological factors to mobile social game addiction. *Telematics Informatics* 33(4):1155–1166.

Chen, K.; Yang, F.; and Chen, X. 2016. Planning with task-oriented knowledge acquisition for a service robot. In *IJCAI*, 812–818. IJCAI/AAAI Press.

Clark, A. B.; Baron, N.; Orr, L.; Kovac, M.; and Rojas, N. 2022. On a balanced delta robot for precise aerial manipulation: Implementation, testing, and lessons for future designs. In *IROS*, 7359–7366. IEEE.

Erdem, E., and Patoglu, V. 2018. Applications of ASP in robotics. *Künstliche Intell.* 32(2-3):143–149.

Erdem, E.; Gelfond, M.; and Leone, N. 2016. Applications of answer set programming. *AI Mag.* 37(3):53–68.

Gebser, M.; Kaminski, R; König, A.; and Schaub, T. 2011. Advances in *gringo* Series 3. In *LPNMR*, volume 6645 of *LNCS*, 345–351. Springer.

Gebser, M.; Liu, L.; Namasivayam, G.; Neumann, A.; Schaub, T.; and Truszczynski, M. 2007. The first answer set programming system competition. In *LPNMR*, volume 4483 of *LNCS*, 3–17. Springer.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub T. 2019. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.* 19(1):27–82.

Gebser, M.; Maratea, M.; and Ricca, F. 2020. The seventh answer set programming competition: Design and results. *Theory Pract. Log. Program.* 20(2):176–204.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *ICLP/SLP*, 1070–1080. MIT Press.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Gener. Comput.* 9(3/4):365–386.

Gielis, K.; Kennes, J.; Dobbeleer, C. D.; Puttemans, S.; and Abeele, V. V. 2019. Collecting digital biomarkers on cognitive health through computer vision and gameplay: an image processing toolkit for card games. In *ICHI*, 1–12. IEEE.

Gualà, L.; Leucci, S.; and Natale, E. 2014. Bejeweled, Candy Crush and other match-three games are (NP-)hard. In *CIG*, 1–8. IEEE.

Hamilton, A.; Nguyen, G. T.; and Roughan, M. 2021. Counting Candy Crush configurations. *Discret. Appl. Math.* 295:47–56.

Ianni, G.; Pacenza, F.; and Zangari, J. 2020. Incremental maintenance of overgrounded logic programs with tailored simplifications. *Theory Pract. Log. Program.* 20(5):719–734.

Ito, T.; Kawahara, J.; Minato, S.; Otachi, Y.; Saitoh, T.; Suzuki, A.; Uehara, R.; Uno, T.; Yamanaka, K.; and Yoshinaka, R. 2022. Sorting balls and water: Equivalence and computational complexity. In *FUN*, volume 226 of *LIPIcs*, 16:1–16:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Kansal, S., and Mukherjee, S. 2022. Kinematic and dynamic analysis of a dexterous multi-fingered delta robot for object catching. *Robotica* 40(8):2878–2908.

Maraj, C. S.; Hurter, J.; and Murphy, S. 2020. Performance, simulator sickness, and immersion of a ball-sorting task in virtual and augmented realities. In *HCI (10)*, volume 12190 of *LNCS*, 522–534. Springer.

Meli, D.; Sridharan, M.; and Fiorini, P. 2021. Inductive learning of answer set programs for autonomous surgical task planning. *Mach. Learn.* 110(7):1739–1763.

Moustris, G. P., and Tzafestas, C. S. 2022. Modelling and analysis of a parallel double delta mechanism for robotic surgery. In *MED*, 861–866. IEEE.

Murphy, R. R. 2000. *Introduction to AI Robotics*. MIT Press.

Quigley, M.; Conley, K.; Gerkey, B.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; Ng, A. Y.; et al. 2009. ROS: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, 5. Kobe, Japan.

Renz, J.; Ge, X.; Gould, S.; and Zhang, P. 2015. The Angry Birds AI competition. *AI Mag.* 36(2):85–87.

Ruff, B. P. D. 1987. A pipelined architecture for the Canny edge detector. In *Alvey Vision Conference*, 1–4. Alvey Vision Club.

Saribatur, Z. G.; Patoglu, V.; and Erdem, E. 2019. Finding optimal feasible global plans for multiple teams of heterogeneous robots using hybrid reasoning: an application to cognitive factories. 43(1):213–238.

Scher, S.; Crabb, R.; and Davis, J. 2008. Making real games virtual: Tracking board game pieces. In *ICPR*, 1–4. IEEE Computer Society.

Song, J., and Li, S. 2018. A robust algorithm for Go image recognition in Go game. In *2018 IEEE 4th International Conference on Computer and Communications (ICCC)*, 1397–1401.

Stanescu, M., and Certický, M. 2016. Predicting opponent's production in real-time strategy games with answer set programming. *IEEE Trans. Comput. Intell. AI Games* 8(1):89–94.

Temel, F. Z.; Doshi, N.; Koh, J.; and Wood, R. J. 2018. The millidelta: A high-bandwidth, high-precision, millimeter-scale delta robot. *Sci. Robotics* 3(14).

Thielscher, M. 2009. Answer set programming for single-player games in general game playing. In *ICLP*, volume 5649 of *LNCS*, 327–341. Springer.

Vischer, P., and Clavel, R. 1998. Kinematic calibration of the parallel delta robot. *Robotica* 16(2):207–218.

Wei, Y.; Huang, T.; Chen, H.; and Liu, J. 2017. Chess recognition from a single depth image. In *ICME*, 931–936. IEEE Computer Society.

Wölflein, G., and Arandjelovic, O. 2021. Determining chess game state from an image. *J. Imaging* 7(6):94.

Yang, H.; Chen, L.; Ma, Z.; Chen, M.; Zhong, Y.; Deng, F.; and Li, M. 2021. Computer vision-based high-quality tea automatic plucking robot using delta parallel manipulator. *Comput. Electron. Agric.* 181:105946.

Yuen, H. K.; Princen, J.; Illingworth, J.; and Kittler, J. 1989. A comparative study of Hough transform methods for circle finding. In *Alvey Vision Conference*, 1–6. Alvey Vision Club.