SAPIENZA
UNIVERSITÀ DI ROMA

HRI & RA Project

Slide Puzzle Game
An interactive social game with Pepper

Professors:
Iocchi
Patrizi

Students:
Bayoumi 1747042
Bellucci 1760390
Di Rienzo 1844531

Academic Year 2022/2023

Contents

Abstract

Puzzle Games represent an interactive and social gaming experience involving active participation from both the user and the social robot Pepper. This game is played in a collaborative way, with the robot and the user working together to reach a final solution. Each turn presents the user with four moves to make, while Pepper two. Throughout the gaming session, there is a constant social interaction between Pepper and the user, occurring before, during, and after gameplay.

The reasoning aspect of this interactive experience is facilitated through the utilization of the AIPlan4EU framework. To ensure efficient communication between the robot and the user, several client and server components have been developed.

The authors equally contributed to the project

1 Introduction

Robots are becoming an increasingly integral part of our society. In recent years, significant importance has been given to communication between robots and humans. The ability to interact with a robot has always been at the forefront of scientific studies, and for this reason, social robots have become increasingly important to us.

Social robots are artificial intelligence platforms paired with sensors, cameras, microphones, and other technologies, such as computer vision, so they can better interact and engage with humans or other robots.

Social robots can be used for various purposes, including education, games, and entertainment. In particular, social robots have been very successful in performing video game-related activities.

One of the most famous social robots is Pepper. Pepper can engage in conversations, learn from the reactions of its interlocutors, respond to their different emotions, and adapt its tone of voice and body language to the context in which it finds itself. This is made possible by a series of functionalities with which it is equipped:

- A set of predefined animations.
- Animated dialogues, allowing Pepper to animate itself automatically when speaking.
- Basic awareness, enabling Pepper to analyze its surrounding environment and react accordingly. Pepper has been enhanced with advanced computer vision functions, including object detection and object recognition.
- Three vocal tones: neutral, happy, and didactic.
- The ability to integrate internal systems with Natural Language Processing engines, facilitating dialogue management with the user.

For this project, *Pepper* (Figure 1) was utilized with the primary objective of enabling interactions with users during a game called "*Puzzle Tiles*". Its role is to engage with the user, teach them how to play the game, and collaborate with the user to jointly find a solution to complete the game level.

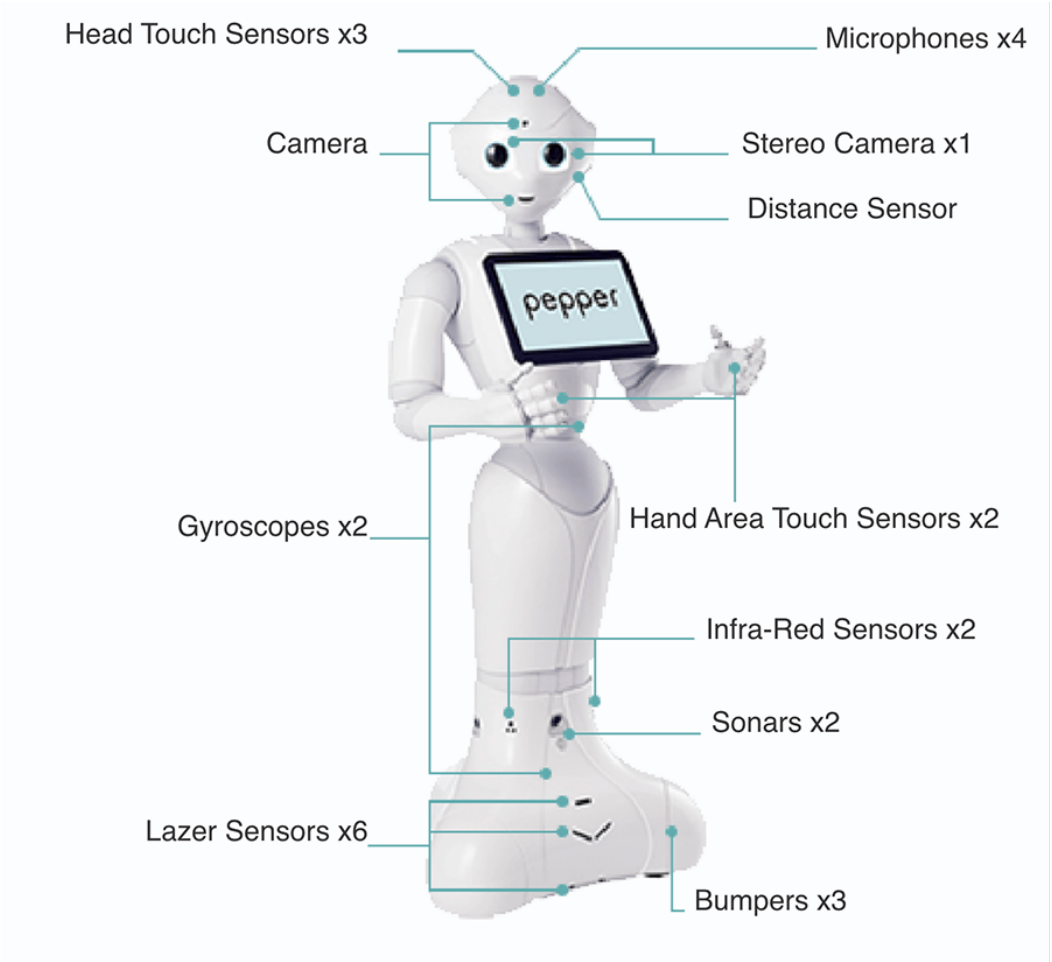


Figure 1: Social robot Pepper with its components.

2 Related Works

Researchers have recently explored the role of social robots in a variety of domains, including shop-keeping [1] [2], education [3], travel [4] and many others. Among them, the **social robots** are gaining an important success in people's life as they are designed to interact with people in a natural, interpersonal manner – often to achieve social-emotional goals in diverse applications such as education, health, quality of life, entertainment, communication, and collaboration.

The long-term goal of creating social robots that are competent and capable partners for people is quite a challenging task. They will need to be able to communicate naturally with people using both verbal and nonverbal signals. They will need also a wide range of social-cognitive skills and a theory of other minds to understand human behavior, and to be intuitively understood by people. Robots are very useful also in solving games together with humans: Games have been used in past human-robot interaction works to investigate robots' social attributes, such as teambuilding [5], facilitating conversations [6], and robot trust [7]. The individual and collaborative cognitive skills that are involved in order to accomplish this task are: geometric reasoning and situation assessment based on perspective-taking and affordance analysis; acquisition and representation of knowledge models for multiple agents

(humans and robots, with their specificities); situated, natural and multi-modal dialogue; human-aware task planning; human–robot joint task achievement and so on.

2.1 Human Robot Interaction

Human–Robot Interaction (HRI) represents a challenge for Artificial Intelligence (AI). It lays at the crossroad of many subdomains of AI and, in effect, it calls for their integration: modelling humans and human cognition; acquiring, representing, manipulating in a tractable way abstract knowledge at the human level; reasoning on this knowledge to make decisions; eventually instantiating those decisions into physical actions both legible to and in coordination with humans. In Lemaignan et al. [8], the main focus is on a specific class of interactions: human–robot collaborative task achievement supported by multi-modal communication. The whole process involved in the study is depicted in Figure 2. In this figure, the human and robot share a common space and are able to socialize through different modalities (gestures and verbal communication). The robot is expected to perform a task where it has to manipulate, fetch and carry objects while taking into account the intentions, beliefs and skills of its human partner and so behave accordingly through planning and proposing resulting plans to the human. The robot controller in fact decides what action to do next (with the help of a task planner) and also who should perform it and what could be done best to facilitate human-robot joint action until the goal is achieved. This involves several planning and decisional skills that are needed to be available to the robot [9].
What is described in this study is what our project aims at doing, so finding an interactive approach in which both robot and human have to plan the minimal sequence of moves to solve the Slide Puzzle game using the reasoning of the robot that helps find the best solution, while instantiating a communication and a social bound with the human.

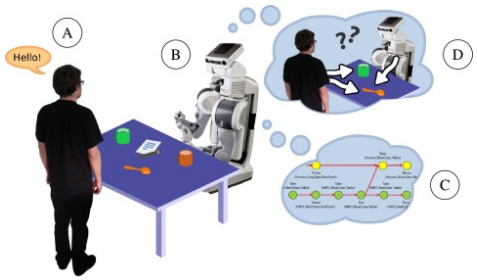


Figure 2: Process of a robot reasoning and cating in domestic interaction scenarios.

2.2 Reasoning Agents

A *sliding puzzle* is a combination puzzle where a player slides pieces along certain routes on a board to reach a certain end-configuration. Solving this type of game is a challenge for the reaserchers who try to find the best approach that reduces the time to reach the final goal and the needed moves, while remaining coherent with the rules of the game. The most used approach in this field is the $A*$ algorithm [10], which is widely used in pathfinding. The key feature of the A* algorithm is that it keeps track of each visited node, which helps in ignoring the nodes that are already visited, saving a huge amount of time. The next node is chosen

on the basis of a combination of heuristic value *h-score* that defines how far the goal node, and the *g-score* which is the number of nodes traversed from the start node to the current node. In a game as the sliding puzzle, the h-score represents the number of misplaced tiles comparing the current state and the goal state, while the g-score will remain as the number of nodes traversed from a start node to the current node.
The idea that is developed in this project is to use a planning algorithm to solve the game. Zhou et al. [11] provide a logic-based multi-paradigm programming language called *Picat* that provides pattern matching, deterministic and non-deterministic rules, loops, list comprehensions, functions, constraints, and tabling as its core modeling and solving features. In this paper, this algorithm is used in order to solve the 15-puzzle game: the game's state is represented as a list of sixteen elements (row, col) that describe the positions of the tiles on the board. The planner module of Picat provides predicates that can be used in order to solve planning problems, such as $final(S)$ that succeds if S is a final state, $action(S, NextS, Action, ActionCost)$ which encodes the set of actions to go from a state to the other and $plan(S, Limit, Plan, PlanCost)$ that builds a plan to transform state S to the final state.
The idea is represented also in our project, where a PDDL-based planning is performed using *AIPlan4EU* (`https://www.aiplan4eu-project.eu/`) that allows to use PDDL planners for our purposes.

3 Solution

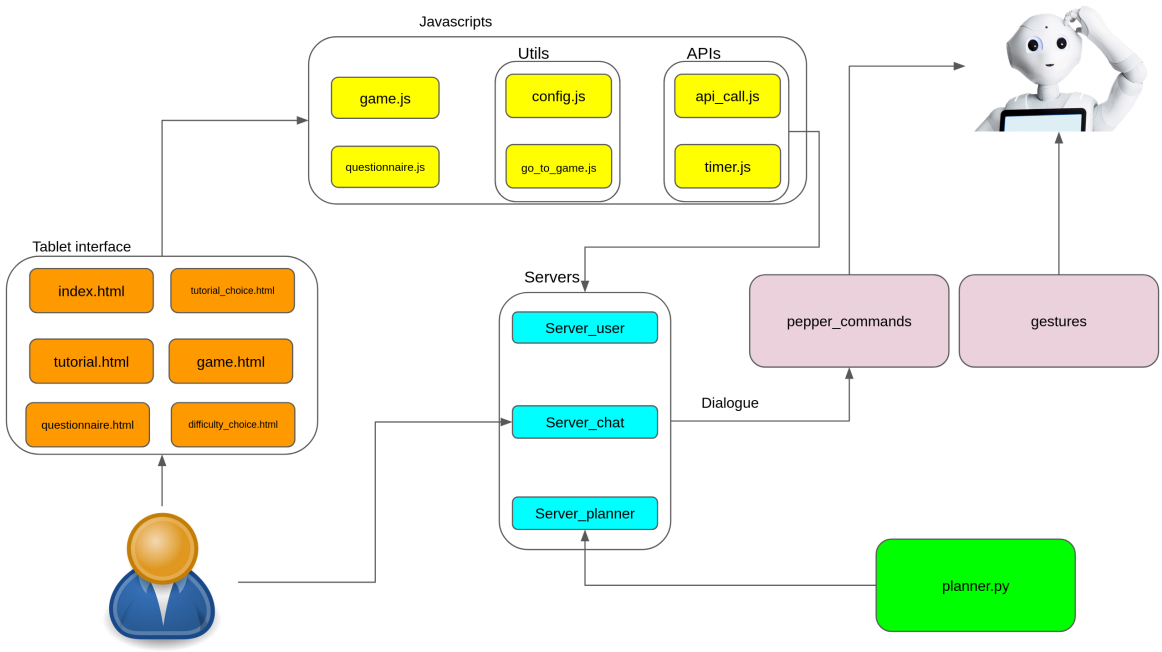


Figure 3: Architecture of Pepper-human interaction.

Figure 3 shows the main components of the architecture proposed as a solution. The interaction between the human and Pepper is possible thanks to the creation of a server that contains three classes:

- *server_chat.py*: This file creates a chat that allows the user answer to questions made by the robot simulating a real interaction.
- *server_planner.py*: It connects the robot to the planner that generates the optimal (or suboptimal) moves to reach the final state.
- *server_user.py*: This file contains functions that keep track of the actual user that is interacting with pepper, in order to collect information about the user that will be needed for future social contacts.

The server is very useful because it enables the direct access to the *pepper_commands* file, which contains all the classes needed to make pepper "alive" (motion and speech). Gestures in particular are managed in the *gestures* file that contains new gestures that can be simulated throughout the interaction.
Pepper's tablet is simulated through the files that are grouped in *Tablet interface*; through HTML files, the user can access to the tutorial of the game and to the game itself, following the guide introduced by pepper at each step.

3.1 Human Robot Interaction

The complete flow diagram of human-pepper interaction is shown in Figure 4. Pepper initially scans the environment to look for humans, and once it finds one, it begins to approach and then scans the human. It introduces itself and asks for the human's name. After that, it asks if the human wants to play with it. If the answer is 'no', Pepper says, 'Okay, sorry to have bothered you', and the process ends. Otherwise, Pepper asks the person to provide his/her name. If the name is already in the database, Pepper says, 'Glad to see you again', and directs the person to the difficulty selection page.

If the user is not registered, Pepper asks 'Do you like puzzle games?' and waits for a response: If the user responds with 'yes', Pepper reacts positively by dancing and saying 'Perfect'. If the person responds with 'no', Pepper responds thoughtfully by saying, 'Hmm... maybe, I'll try to change your mind.' After this step, Pepper wants to know if the unregistered user has previously played the "Slide Puzzle" game with Pepper. The user has to provide an answer and if he/her has already played the game with Pepper, the difficulty selection page open, where Pepper lists the available choices. In the other case, Pepper starts to explain the rules to the user. After completing the tutorial, an unregistered user will start a new game at the easiest difficulty level being the first time. In the case where the user in an old one, Pepper will remind the last played difficulty and his/her preference.

At the end of the game, the robot asks if the user wants to play again. When the user has finished the experience of playing the game, the robot says, 'Before we say goodbye, I would like you to fill in this survey to get your opinion', and directs the user to the questionnaire page. Once the questionnaire is completed and submitted, Pepper says 'Nice to meet you, see you next time' for unregistered users and 'Nice to see you again, see you next time' for registered users, leading to a storage or update in the database of the preferences of the user selected in the questionnaire page. In both cases, after saying the sentence, the process ends.

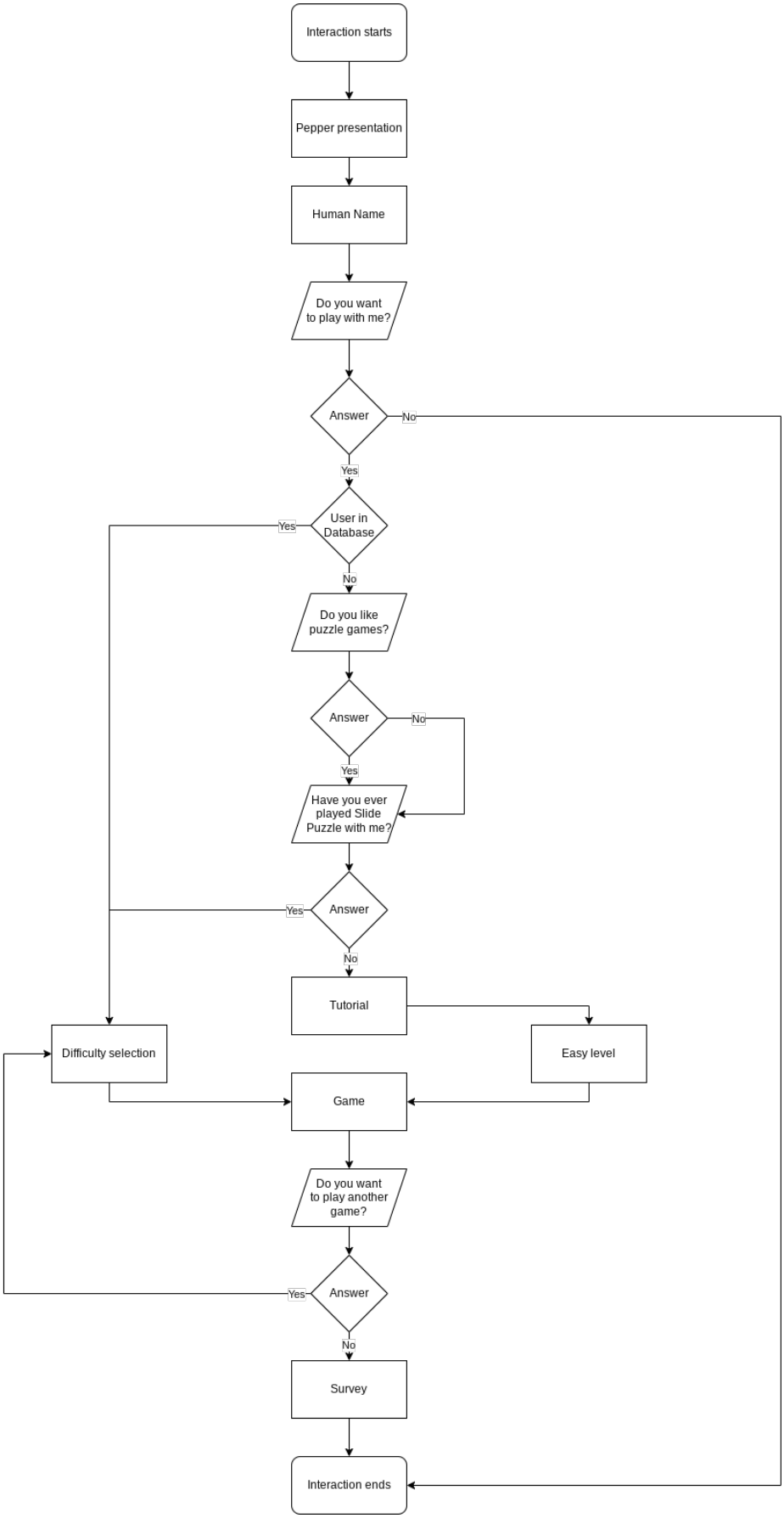
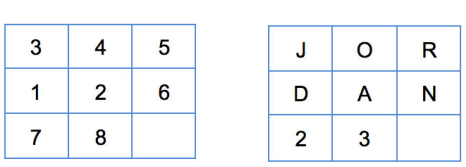


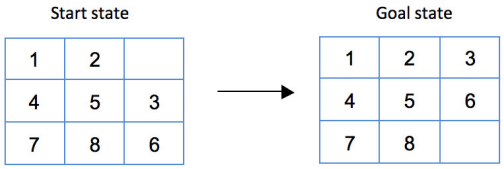
Figure 4: Flow diagram HRI.

3.2 Reasoning Agents

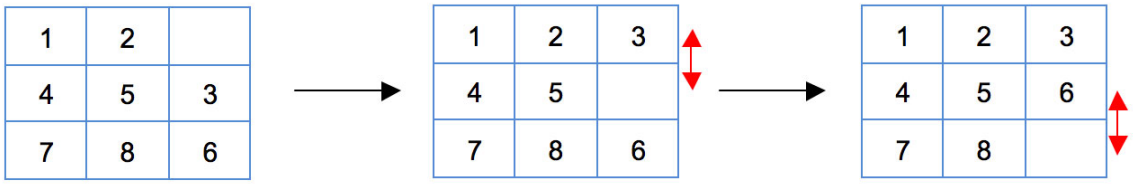
The *Sliding Tiles Puzzle* was created by chess player and puzzle maker Sam Loyd in the 1870s. In Figure 5a is shown how a puzzle NxM is built, where each cell could represent a number, a letter, a patch of an image (like oour project) and so on. The task that has to be accomplished is to reach a goal state (another configuration) from a state where the tiles are randomly distributed along the board, so rearrange the tiles in order to match the final configuration (Figure 5b). In order to reach the goal state, a sequence of moves has to be performed by swapping the empty tile with some other tile in the near (Figure 5c).



(a) 3x3 sliding Tiles Puzzles Board.



(b) moving tiles from start state to goal state.



(c) Swap tiles examples.

Figure 5: Sliding Tiles Puzzle.

In order to model the game presented, the idea is to create a matrix NxM (where N and M are different for each level of difficulty chosen) that contains the positions of each tile, where tiles are signed with a number from 0 to number of tiles - 1, and the last tile will be the empty one. So each tile's position is denoted by a couple (i, j) where i denotes the row and j the column. The "slide" is possible when swapping positions of the empty tile and the adjacent tiles. The final state goal is composed by number of the tiles that are correctly ordered, with the empty tile in the bottomright position (as in Figure 5).

4 Implementation

4.1 Experimental Setup

The program the simulates the behavior of Pepper is *Choreographe*[1] that is able to run a simulation of Pepper robot and NAO Robot. It is connected to our code through an IP and Port where the robot runs and it is controlled thanks to the use of *pepper_tools*, which is a repository that contains different tools to access robots functions. The main language that is used to implement our code is Python, while for creating the tablet interface we have used Javascript, HTML and CSS.

[1] `http://doc.aldebaran.com/1-14/software/choregraphe/choregraphe_overview.html`

We have generated a server through the *Flask* [2]framework that contains tools for a Web server. In order to maintain an exchange of data between the robot and the user and between the tablet and the robot, we have integrated the Flask server with *Rest APIs* in Python using the Restful extension of Flask. Flask RESTful defines a class called *Resource* which is the base class for all the resources. When adding a new Resource class, it will be needed an *endpoint* which is a service that gives a client the necessary information to gain access to the resources. Each resource can have several methods associated with it such as GET (return a representation of a resourse), POST (create a resourse), PUT (update a resorse), DELETE (remove a resourse), etc. Those methods are useful for our approach because they make it possible to transfer data from the server and so to continuously control robot's state and the game status. In order to keep track of the information of an user during the interaction (like his/her name, record, games won and many others), JSON files are created in which these data are stored and then used in future to make Pepper recognize the user.

4.2 Human-Robot Interaction

This section describes the classes and the relative functions used to communicate with Pepper and its movements.

4.2.1 Pepper Commands

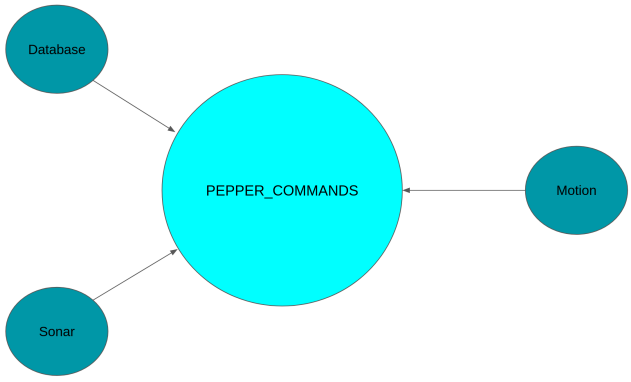


Figure 6: Architecture of Pepper-human interaction.

As mentioned before, *pepper_tools* allow to manage Pepper robot through functions that make use of the modules in Pepper (like *ALTextToSpeech*, *ALMemory*, *ALMotion*...). These modules control Pepper's sentences, movements and sensors. In the following, the implemented classes are shown.
The first task that Pepper does is to search for humans in the near, and this is done through the *Sonar()* and *Motion()* classes. The Sonar() takes into account the values of the virtual sonar sensors (in this case, only the frontal one to detect humans in front of the robot) and it calculates the distance from the robot to the human detected. After having found

[2]`https://flask.palletsprojects.com/en/2.3.x/`

the human, the Motion() class controls the values returned by the sonar sensors in order to select the minimum distance required to reach the human (setting the speed).

```
class Motion:
    def __init__(self):
        self.motion_service = pepper_cmd.robot.motion_service

    def setSpeed(self, lin_vel, ang_vel, dtime, sonar):
        self.motion_service.move(lin_vel, 0, ang_vel)
        time.sleep(dtime)
        self.motion_service.stopMove()
        return

    def forward(self, sonar, s, lin_vel=0.2, ang_vel=0):
        print("Sonar Values", sonar.sonarValues)
        self.setSpeed(lin_vel, ang_vel, abs((s-0.5)/lin_vel), sonar)
        # update the new position of the robot

    def detect_person(self, sonar):
        for i in range(len(sonar.sonarValues)):
            if sonar.sonarValues[i] <= 0.75:
                if i==0:
                    print("Person detected with sonar SonarFront")
                else:
                    print("Person detected with sonar SonarBack")
                return True
        return False

    def selectMinDistance(self, distances):
        min_distance = float("inf")
        index = 0

        for i in range(len(distances)):
            if distances[i] < min_distance:
                min_distance = distances[i]
                index = i

        return min_distance, index
```

```
class Sonar:
    def __init__(self, robot_position, sensor= "SonarFront", duration = 3.0):
        self.sensor = sensor
        self.memory_service = pepper_cmd.robot.memory_service
        self.duration = duration
        self.sonarValueList = ['Device/SubDeviceList/Platform/Front/Sonar/Sensor/Value',
                               'Device/SubDeviceList/Platform/Back/Sonar/Sensor/Value' ]
        self.robot_position = robot_position # tuple (int, int)
        self.humans_positions = self.get_positions()

    def set_sonar(self):
        distances = self.get_distances()
        mkey = self.sonarValueList[0]
        self.memory_service.insertData(mkey,distances)
```

```
        mkey = self.sonarValueList[1]
        self.memory_service.insertData(mkey,None) #disabled
        time.sleep(self.duration)
        self.sonarValues =  self.memory_service.getListData(self.sonarValueList)
        print(self.sonarValues)

    #we use only the frontal sonar, so it will discover only humans in front of him
    def get_positions(self):
        human1_position = (1,0)
        human2_position = (2,0)
        humans_positions = [human1_position, human2_position]

        return humans_positions

    def get_distances(self):
        distances = []
        for pos in self.humans_positions:
            distance = math.sqrt((self.robot_position[0]-pos[0])**2 +
            ↪  (self.robot_position[1]-pos[1])**2)
            distances.append(distance)

        return distances
```

The information is initialized through the Database() class, which manages the new and old users as it follows.
When a new user is encountered, Pepper searches for the name in its database (a json file named *registered_users*) and acts consequently:

- If the user has never been met before, Pepper welcomes the user and the name is added to the registered_users and the correlated values needed are initialized.
- Otherwise Pepper recognizes the user and asks which level of the game he wants to choose and the game starts. In this case, Pepper tells also the user which was the favourite level (the answer of the final survey) and the last level played.

The information stored in the json file are statistics about the difficulty level of the games played by the user, lik *record_moves* (minimum moves to complete the level), *record_time* (minimum time elapsed) and *num_games_won* (number of games completed). In addition, the *last_difficulty* and *Survey* values are stored, where the first is the last level difficulty played and the other corresponds to the answers of the final survey.

```
class Database:
    def __init__(self, filename):
        self.file = "./data/" + filename + ".json"
        self.chat = Dialogue()
        self.is_new = False

    def create_db(self):
        if not os.path.exists(self.file):
            with open(self.file, 'w') as f:
                json.dump({}, f, indent=4)
```

```
11
12          with open("./data/game_pepper_interaction.json", 'w') as f:
13              json.dump({
14                  "win": False,
15                  "robot_moves": [],
16                  "interaction": ""
17              }, f, indent=4)
18
19      def name_user(self):
20          name = self.chat.say("What is your name?"+" "*5, require_answer=True)
21          self.save_user(name)
22          print("Name is "+ name)
23
24      def save_user(self, name):
25          with open("./data/actual_user.json", 'w') as f:
26              json.dump({"user": name}, f, indent=4)
27
28      def detect_user(self):
29          data = {}
30          if os.path.exists(self.file):
31              name = self.name_user()
32              with open(self.file, 'r') as f:
33                  data = json.load(f)
34
35                  if name in data.keys(): #already registered user
36                      print("{} exists in the database".format(name))
37                      self.chat.say("Glad to see you again {}!".format(name))
38                  else:
39                      print("{} not exists in the database".format(name))
40                      self.chat.say("Nice to meet you {}!".format(name))
41                      self.is_new = True
42
43              if self.is_new:
44                  self.register_user(data, name)
45
46          return name
47
48      def register_user(self, data, name):
49          if os.path.exists(self.file):
50              with open(self.file, 'w') as f:
51
52                  data[name] = {
53                      "Games": {
54                          "easy": {
55                              "record_moves": "-",
56                              "record_time": "--:--",
57                              "num_games_won": 0
58                          },
59                          "medium": {
60                              "record_moves": "-",
61                              "record_time": "--:--",
62                              "num_games_won": 0
63                          },
```

```
                        "hard": {
                            "record_moves": "-",
                            "record_time": "--:--",
                            "num_games_won": 0
                        },
                        "last_difficulty": "easy"
                    },
                    "Survey": {}
                } #create an entry with user data
                json.dump(data, f, indent=4)
        else:
            print("No Database has been created")

```

The most important feature that is highlighted in this project is the interaction between the human and the robot, and the verbal communication is at its base. In order to make the robot speak, the class *Dialogue()* (stored in the file *chat.py* in *utils* folder) uses the pepper command tool *say*. The *say* function operates with the ALTextToSpeech module that enables the robots say something. Being Choreographe a simulation environment, the sentences are shown as a text message above the robot. This class makes a difference between when the robot has to say something and when it expects that the human answers. If the human does not respond within 15 seconds, it says *"Sorry, I did not hear you, repeat please."* and waits for the answer again (function *listen*). The function *listen_multi* is implemented in order to make the human answer only between *"yes"* or *"no"*.

```
class Dialogue:
    def __init__(self):
        self.api = API_call(URL, "chat")

    def say(self, sentence, require_answer = False, choices = []):
        pepper_cmd.robot.say(sentence + " "*15)

        if require_answer:
            if choices == []:
                return self.listen(sentence)
            else:
                return self.listen_multi(sentence, choices)
        else:
            self.api.call('get',  TIMEOUT, ['req', GET_MSG], ['sentence',sentence]) #pepper
            ↪  sentence


    def listen(self, sentence):
        try:
            answer = self.api.call('get',  TIMEOUT, ['req', GET_ANS],
            ↪  ['sentence',sentence])["response"] #human answer
        except:
            answer = self.say(sentence = "Sorry, I did not hear you, repeat please.",
            ↪  require_answer = True)
        return answer

    def listen_multi(self, sentence, choices):
```

```
25          answer = self.api.call('get',  9000, ['req', GET_MULTI], ['sentence',sentence],
            ↪  ["choices", ",".join(choices)])["response"] #human answer
26          if answer.lower() not in choices:
27              answer = self.say(sentence = "Sorry I didn't understand, can you repeat
                ↪  please?", require_answer=True, choices=choices)
28          return answer
```

4.2.2 Gestures

Among the functions that interact with pepper, there are some created in order to make the robot move and react to what happens.
These functions are implemented in the *Gestures()* class.
The gestures developed are:

- *doHello()*: greets the user.
- *movetileRight(), movetileLeft(), movetileUp(), movetileDown()*: Makes a gesture that moves the tile in the chosen direction.
- *gestureSearching()*: Moves the head right and left to simulate a searching gesture.
- *gestureAnalyzing()*: Moves the head up and down to simulate a scanning gesture.
- *doYes()*: Moves the head up and down to simulate a positive feedback when the user makes a good move.
- *doNo()*: Moves the head right and left to simulate a negative feedback when the user makes a move that the robot does not agree with.
- *doRock()*: Simulates the play of a guitar.
- *getThinkingPose()*: Pepper simulates a thinking pose when it has to think about the next move to do.
- *doWin()*: When the user wins the game, the robot simulates a win gesture.

In order to create a gesture, the *ALMotion* module is used to change the angles of the various joints, like shown in the example code below.

```
1  def movetileRight(self):
2      isAbsolute = True
3      # move posture
4      jointNames = ["RShoulderPitch", "RShoulderRoll", "RElbowYaw", "RElbowRoll", "RWristYaw",
       ↪  "RHand"]
5      jointValues = [1.57, -0.33, 1.75, 1.23, -0.1, 0.70]
6      times = [0.8, 0.8, 0.8, 0.8, 0.8, 0.8]
7      self.ALMotion.angleInterpolation(jointNames, jointValues, times, isAbsolute)
8
9      # # arm that slides
10     jointNames = ["RElbowRoll", "RElbowYaw", "RShoulderRoll", "RWristYaw"]
11     jointValues = [1.38, 1.23, -0.25, -0]
12     times = [0.6, 0.6, 0.6, 0.6]
```

```
13      self.ALMotion.angleInterpolation(jointNames, jointValues, times, isAbsolute)
14
15  def doYes(self):
16      yes_thread = threading.Thread(target=self.chat.say,
        ↪  args=(random.choice(self.yes_sentence),))
17      yes_thread.start()
18      #move head up and down
19      for _ in range(2):
20          jointNames = ["HeadPitch"]
21          angles = [-0.3]
22          times  = [1.0]
23          isAbsolute = True
24          self.ALMotion.angleInterpolation(jointNames, angles, times, isAbsolute)
25
26          jointNames = ["HeadPitch"]
27          angles = [0.1]
28          times  = [1.0]
29          isAbsolute = True
30          self.ALMotion.angleInterpolation(jointNames, angles, times, isAbsolute)
```

4.2.3 HRI - Servers

As Section 3 points out, the communication through the various parts of our project is possible thanks to the implementation of more servers that drive the exchange of data.
In order to provide a trade-off of information that are needed during the chat and the game, we have created two servers:

- *Server user*: Stores information about the actual user that is playing (in a json file called *actual_user*) and it has two api *post* and *get* in which the first is useful to save the answers of the survey that is submitted and the second picks the name.

```
1   def post(self):
2           self.name = request.args.get("name")
3           json_path = request.args.get("json_path")
4           self.survey = request.json
5           if self.req == POST_SURVEY:
6               with open("./data/registered_users.json", 'r') as f:
7                   data = json.load(f)
8               data[self.name]["Survey"] = self.survey
9               with open("./data/registered_users.json", 'w') as f:
10                  json.dump(data, f, indent=4)
11              return {"message": "User preferences modified", "error": False}
12          return {"message": "POST request failed", "error": True}
13
14  def get(self):
15          if self.req == GET_USER:
16              with open("./data/actual_user.json", 'r') as f:
17                  data = json.load(f)
18              return {"message": "User returned", "error": False, "response": data}
19
20          return {"message": "GET request failed", "error": True}
```

- *Server chat*: This server allows the user to generate an answer to Pepper's questions, and those answers are controlled in the *get* api. This api takes the sentences returned from Pepper and gives in output what the human says. The get api is also useful to open the web interface when needed.

```
def get(self):
    if self.req == GET_ANS:
        print("Pepper says: ", self.sentence)
        try:
            nome = inputimeout("Human answer: ", timeout=TIMEOUT)
        except Exception:
            return {"message": str(Exception), "error": True}
        return {"message": "Answer returned", "error": False, "response": nome}
    if self.req == GET_MSG:
        print("Pepper says: ", self.sentence)
        return {"message": "Pepper message returned", "error": False}
    if self.req == GET_MULTI:
        print("Pepper says: ", self.sentence, self.choices.split(","))
        try:
            answer = inputimeout("Human answer: ", timeout=9000)
        except Exception:
            return {"message": str(Exception), "error": True}
        return {"message": "Answer returned", "error": False, "response": answer}
    if self.req == GET_HTML:
        subprocess.check_output("firefox " + self.page, shell=True,
        ↪  universal_newlines=True)
        return {"message": "Pepper message returned", "error": False}

```

4.3 Web interface

In this section, we introduce the web pages we developed in order to recreate the display of Pepper's tablet.

4.3.1 select_difficulty.html

Figure 7: Html page for difficulty selection.

This page (Figure 7) provides users with the option to choose their preferred difficulty level. Additionally, Pepper interacts with the user in two specific scenarios described in

```
async function read_num_game_info() {
    const api_client = new ApiClient(URL_BASE)

    var queryParams = [
        ["req", GET_JSON],
        ["json_path", "./data/registered_users.json"]
    ]
    let registered_users;
    await api_client.get('/planner', queryParams)
        .then(data => {
            registered_users = data.response;
        })
        .catch(error => console.error(error));
    queryParams = [
        ["req", GET_JSON],
        ["json_path", "./data/actual_user.json"]
    ]
    let actual_user;
    await api_client.get('/planner', queryParams)
        .then(data => {
            actual_user = data.response.user;
        })
        .catch(error => console.error(error));
    var user_game_info = registered_users[actual_user]
    var n_easy = user_game_info.Games.easy.num_games_won
    var n_medium = user_game_info.Games.medium.num_games_won
    var n_hard = user_game_info.Games.hard.num_games_won

    document.getElementById("easy").textContent = "Total win: " + n_easy
    document.getElementById("medium").textContent = "Total win: " + n_medium
    document.getElementById("hard").textContent = "Total win: " + n_hard
    console.log(n_easy, n_medium, n_hard)
}
```

Listing 1: Difficulty selection script.

Section 5.3. The buttons if clicked will lead to the game to the various difficulties through Script 1. The web page keeps track of the total wins achieved by the user for each level.

4.3.2 tutorial.html

This page (Figure 8) serves as an interactive tutorial for the game, with Pepper taking on the role of an instructor. During this stage, Pepper provides explanations about the various elements present on the page, helping the user understand the available actions and moves he can make. The clickable tiles, are highlighted in green.

After completing the tutorial, the user will be automatically redirected to the HTML page of the game, set at the easy difficulty level.

The tutorial level, unlike the actual game, is not randomly generated but is done by hand to be as simple as possible and make the user better understand how the game works.

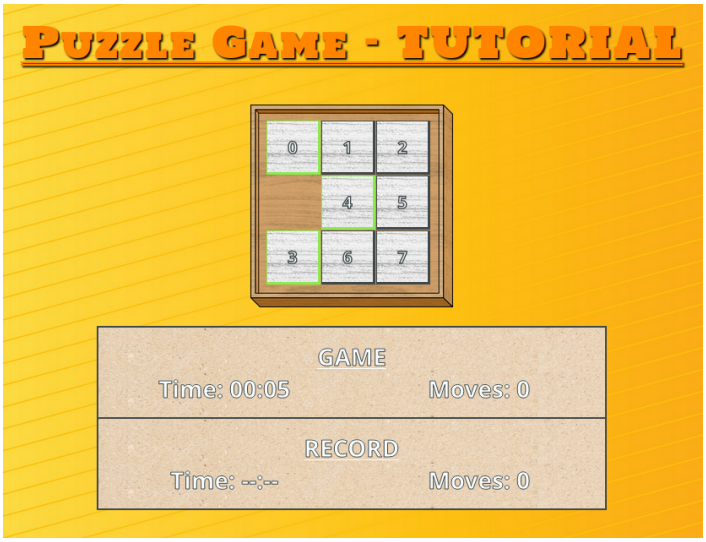


Figure 8: Html page of the tutorial

4.3.3 game.html

(a) Easy. (b) Medium. (c) Hard.

Figure 9: Game levels.

The main game screen consists of a puzzle box with tiles inside, where the number of tiles varies according to the type of level (Figure 9). The image that will be split into tiles is randomly selected each time the game is started. Under the box there is a rectangle containing the number of current moves and the elapsed time since the start of the game, below that there are the user's record in terms of time and moves employeed for that difficulty level. Interaction in the game is done by alternating between four user moves and two Pepper moves. The user can interact with the puzzle box in the same way as he did with the puzzle box in the tutorial level, so possible tiles that can be moved will be highlighted in green. Once the user has made 4 moves the turn is passed to Pepper. From an interface perspective this is handled by making the tiles unclickable. In addition, to coincide Pepper's gestures with moving the tiles within the puzzle box, a sleep timer is added that waits before moving the tiles. In this way the user will have the impression that he is actually playing with Pepper. Once the game has finished, a screen will be shown with the sentence "We won" (Figure 10) and then the user will be asked if he would like to play another game, being then redirected to the difficulty choice page. Otherwise he will be sent to the page for filling out the questionnaire.

Figure 10: Victory page

4.3.4 questionnaire.html

This page presents to the user a form containing various questions related to the game and its interaction with Pepper. Specifically, the questions are the following:

1. How satisfied are you with the overall experience of the cooperative game with the robot?
2. Was the tutorial sufficient for you to understand the game?
3. Which difficulty level did you find most suitable for your skills and preferences?
4. How do you rate the interaction with the robot during the game?
5. Did you encounter any issues in communicating with the robot or understanding its moves?
6. Did you feel the robot was an effective partner in solving the challenges?
7. Did you feel the robot understood and responded to your actions and strategies appropriately?
8. How would you rate the game?
9. How much would you recommend this game to others?
10. How much would you rate the overall experience and interaction with Pepper?

Users can express their satisfaction levels by selecting the appropriate radio buttons. Additionally, all user responses will be recorded within a JSON file for managing future interactions between the robot and the same user.

4.4 Reasoning Agents

4.4.1 Slide Puzzle

```
class Slide_tile():
    def __init__(self, n_row, n_col, n_moves=100):
        self.n_row = n_row
        self.n_col = n_col
        self.tiles, self.b_x, self.b_y = self.__generate_tiles()
        self.shuffle(n_moves)
```

The slide puzzle is managed through the Slide_Tile class, which will generate an n_row by n_col matrix of numbers. The position of the blank tile will be stored in b_x (row) and b_y (column) and in the tiles matrix will be represented as the "-" character. This class is useful for making moves in python in a simple way and creating a problem to solve.

```
def shuffle(self, n_moves=100):
    for i in range(n_moves):
        r = random.randint(0,3)
        if r == 0:
            self.move_up()
        if r == 1:
            self.move_down()
        if r == 2:
            self.move_right()
        if r == 3:
            self.move_left()
    return
```

A game will be created by taking the solution configuration (with the numbers from 0 up to (n_row x n_col) - 2 followed by the blank in the bottom right corner) to which n_moves will be applied randomly via the shuffle method, in order to generate a configuration that is solvable.

```
def move_right(self):
    if self.b_y != 0:
        tmp = self.tiles[self.b_x][self.b_y-1]
        self.tiles[self.b_x][self.b_y-1] = self.tiles[self.b_x][self.b_y]
        self.tiles[self.b_x][self.b_y] = tmp
        self.b_y -= 1
```

As already mentioned a move means swapping an adjacent square with white space. This is how a move to the right is done (move to the left, upper and bottom are the same conceptually but the indices change) in which the cell adjacent to the blank is the one to its left and the move is "move the selected cell to the right". As can be seen, it is first checked that the blank column is not the firston the board, otherwise there would be no cell to move to its left, then a classic swap is performed and the blank column is updated.

4.4.2 AIPlan4EU implementation

Pepper's reasoning capabilities are provided and implemented through the *AIPlan4EU Unified Planning* framework that exploits the Unified Planning library. Different planners such as *pyperplan*, *tamer*, *enhsp*, *fast-downward* are present. The one used is *fast-downward*. This one was chosen because, unlike the others, it finds optimal solutions very often and supports timeout. Timeout is used when the optimal plan is too slow to be found, in which case a satisfactory plan would be calculated.

Three difficulties are used:

- *Easy*. The board consists of a 3 x 3 cell matrix and the number of random moves to initialise the level is 100

- *Medium.* The frame is composed of a 3 x 4 cell matrix and the number of random moves to initialise the level is 80
- *Hard.* The frame is composed of a 4 x 4 cell matrix and the number of random moves to initialise the level is 80

In the case of easy level, the problem returned is the following:

```
GAME = [
    [1 7 4]
    [2 - 5]
    [0 6 3]
]
types = [Tile, Position]

fluents = [
    bool is_tile[t=Tile]
    bool is_position[p=Position]
    bool blank[bx=Position, by=Position]
    bool at[t=Tile, px=Position, py=Position]
    bool dec[p1=Position, p2=Position]
    bool inc[p1=Position, p2=Position]
]

actions = [
    action Move Up(Tile t, Position px, Position py, Position bx) {
        preconditions = [
            is_tile(t)
            is_position(px)
            is_position(py)
            is_position(bx)
            inc(bx, px)
            blank(bx, py)
            at(t, px, py)
        ]
        effects = [
            blank(bx, py) := false
            at(t, px, py) := false
            blank(px, py) := true
            at(t, bx, py) := true
        ]
    }
    action Move Down(Tile t, Position px, Position py, Position bx) {
        preconditions = [
            is_tile(t)
            is_position(px)
            is_position(py)
            is_position(bx)
            dec(bx, px)
            blank(bx, py)
            at(t, px, py)
        ]
        effects = [
```

```
            blank(bx, py) := false
            at(t, px, py) := false
            blank(px, py) := true
            at(t, bx, py) := true
        ]
    }
    action Move Right(Tile t, Position px, Position py, Position by) {
        preconditions = [
            is_tile(t)
            is_position(px)
            is_position(py)
            is_position(by)
            dec(by, py)
            blank(px, by)
            at(t, px, py)
        ]
        effects = [
            blank(px, by) := false
            at(t, px, py) := false
            blank(px, py) := true
            at(t, px, by) := true
        ]
    }
    action Move Left(Tile t, Position px, Position py, Position by) {
        preconditions = [
            is_tile(t)
            is_position(px)
            is_position(py)
            is_position(by)
            inc(by, py)
            blank(px, by)
            at(t, px, py)
        ]
        effects = [
            blank(px, by) := false
            at(t, px, py) := false
            blank(px, py) := true
            at(t, px, by) := true
        ]
    }
]

objects = [
    Tile: [t0, t1, t2, t3, t4, t5, t6, t7]
    Position: [x0, x1, x2, y0, y1, y2]
]

initial fluents default = [
    bool is_tile[t=Tile] := false
    bool is_position[p=Position] := false
    bool blank[bx=Position, by=Position] := false
    bool at[t=Tile, px=Position, py=Position] := false
    bool dec[p1=Position, p2=Position] := false
```

```
    bool inc[p1=Position, p2=Position] := false
]

initial values = [
    is_position(x0) := true
    ...
    is_position(y2) := true
    is_tile(t0) := true
    ...
    is_tile(t7) := true
    inc(x0, x1) := true
    ...
    inc(y1, y2) := true
    dec(y1, y0) := true
    ...
    dec(x2, x1) := true
    at(t1, x0, y0) := true
    ...
    at(t3, x2, y2) := true
    blank(x1, y1) := true
]

goals = [
    at(t0, x0, y0)
    at(t1, x0, y1)
    at(t2, x0, y2)
    at(t3, x1, y0)
    at(t4, x1, y1)
    at(t5, x1, y2)
    at(t6, x2, y0)
    at(t7, x2, y1)
]
```

We have declared two types of objects: Tile and Position. Tile represents a tile within the board and Position a position x (row) y (column). The number of objects depends on the levels.

Fluents are variable in a planning problem and their values may change over time. In our problem we have:

- *is_tile(t)* Checks if object t is a tile.
- *is_position(p)* Checks if object p is a position.
- *blank(bx, by)* Check if there is blank in row bx and column by
- *at(t, px, py)* Check if tile t is present at row px and column py
- *inc(p1, p2)* Check if p1 and p2 positions are adjacent in ascending order vertically or horizontally (n.b. the ascending order for the rows is downwards)
- *dec(p2, p1)* Check if p2 and p1 positions are adjacent in descending order vertically or horizontally (n.b. the descending order for the rows is upwards)

The following actions were used to describe the evolution of the system (An explanation of how the actions are carried out can be seen in the Figure 5):

- *Move Up(t, px, py, bx)* Allows the tile t to be moved up from the px row and py column position only if the bx row position of the blank is vertically adjacent in ascending order to the px column position and then the blank is exactly above the tile.
- *Move Down(t, px, py, bx)* Allows the tile t to be moved down from the px row and py column position only if the bx row position of the blank is vertically adjacent in descending order to the px column position and then the blank is exactly below the tile.
- *Move Right(t, px, py, by)* Allows the tile t to be moved right from the px row and py column position only if the by column position of the blank is horizontally adjacent in descending order to the py column position and then the blank is exactly to the right of the tile.
- *Move Left(t, px, py, by)* Allows the tile t to be moved left from the px row and py column position only if the by column position of the blank is horizontally adjacent in ascending order to the py column position and then the blank is exactly to the left of the tile.

Continuing with the explanation, we add all objects of type Cells and Vehicle to the problem and initialize the fluents by making the "small-world assumption", i.e. just indicate the fluents that are initially true. The goal of the planning problem is to have the fluent *at* with objects *t_1*, ..., *t_n* placed in order from left to right, top to bottom.

```
def solve_problem(self, timeout=False):
    opt = PlanGenerationResultStatus.SOLVED_OPTIMALLY if (self.optimal_plan and not timeout)
    ↪ else None
    with OneshotPlanner(name='fast-downward', optimality_guarantee=opt) as planner:
        if self.optimal_plan and not timeout:
            result = planner.solve(self.problem, timeout=TIMEOUT)
            if result.status == PlanGenerationResultStatus.TIMEOUT:
                self.solve_problem(timeout=True)
                return
        else:
            result = planner.solve(self.problem)
        self.plan = result.plan
```

As can be seen from the function, firstly a search for an optimal plan will be attempted, but if it is not found within a TIMEOUT time (which in our case is 10 seconds) then the function is called back looking for a satisfactory plan. Once the plan is found it is saved.

4.4.3 RA Server

Through the *server planner*, it is possible the communication of the moves hinted by the plan to the html page, via API call in the javascript code of the *game.js* in which they are then done graphically. The choice to have the user make multiple moves is made to give

the impression that the user is playing interactively with the robot, which would otherwise revert the wrong moves. Given this interaction and the use of a deterministic planner makes it impossible to use a single plan. In fact, what happens is that after the user's moves are compared to those of the plan, if they match they are considered as the best moves and case Pepper will cheer. Otherwise, since the original plan can no longer be executed, another one is created from the current state. In this case, if the new generated plan is longer than the previous one then the user's made moves are considered as "bad" and Pepper will get disappointed. On the other hand, it is possible that the new plan is still shorter, obviously longer than the previous one minus the user's moves, and in this case the moves are considered good but not optimal and Pepper will fake thinking about alternatives.

5 Results

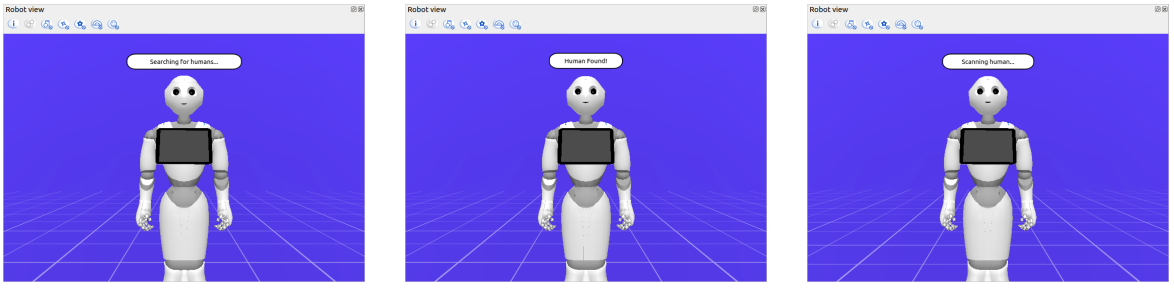
This section focuses on the whole interaction that is established between the robot and the human, and the reasoning skills that are inserted during the gameplay. In this section, both terminal and pepper simulator outputs are presented.

5.1 Pre-Interaction

Before starting the human-robot interaction, Pepper scans the environment and searches for humans within a range using its sonar sensor. When a human is found, Pepper moves up to the human and performs a scan (only computing a gesture) and starts the interaction with the user. Figure 12 shows what Pepper does, while Figure 11 displays on the server what Pepper says.

```
Pepper says:  Searching for humans...
Pepper says:  Human Found!
Pepper says:  Scanning human...
Pepper says:  Human Scanned!
```

Figure 11: Terminal output.



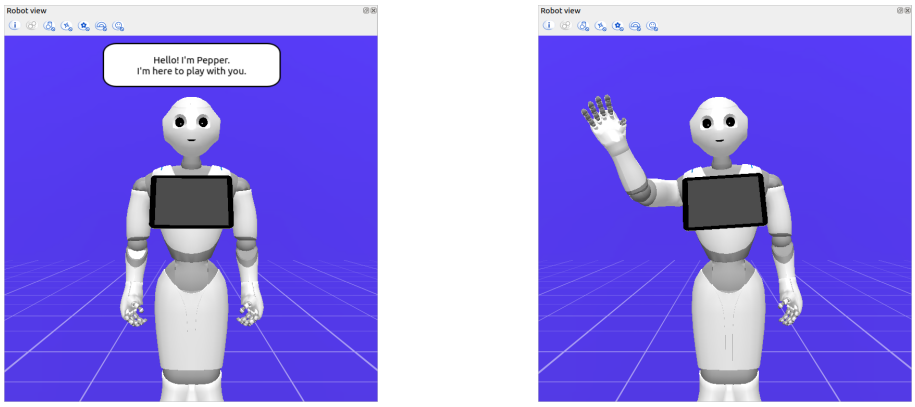
(a) Searching for human. (b) Human found. (c) Scanning the human.

Figure 12: Pre-Interaction.

5.2 Interaction

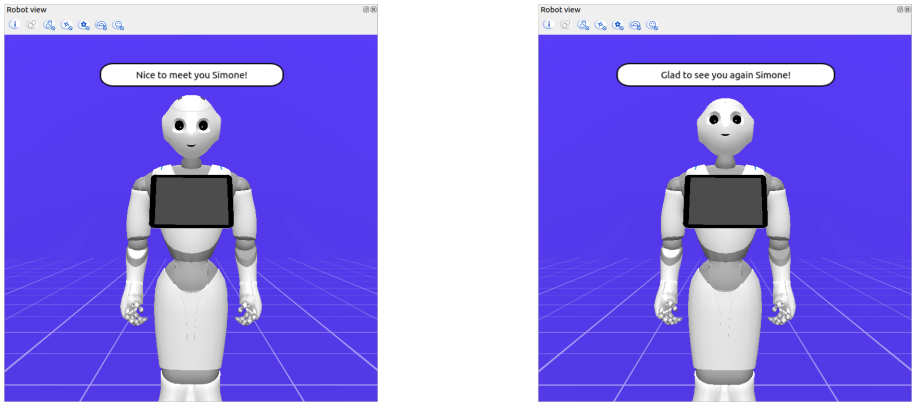
5.2.1 Meet an user

When the robot reaches the human, the front interaction starts with Pepper that introduces itself (Figure 13) and starts to talk with the user by asking his/her name, reacting in two different ways (Figure 14); if it is the first time that Pepper meets the user (Figure 14a, it welcomes him/her friendly. When Pepper has already bumped into the user (after looking at the related database) like in Figure 14b, Pepper greets him/her. In both cases, after Pepper's greetings, the first phase of the game is started and it is described in the section that follows.



(a) Pepper's description. (b) Hello gesture.

Figure 13: Pepper's introduction.



(a) Pepper welcomes the new user. (b) Pepper greets the known user.

Figure 14: Pepper meets an user.

5.3 Game

Before going to the starting page of the game, Pepper asks the user if he/she wants to play a game, and behaves differently accordingly to the answer given. In Figure 16 are indicated the reactions when the user answer either *"yes"* or *"no"* (other answers are not supported and Pepper points it out). Figure 15 shows an example of the question that expects *"yes"* or *"no"* and the relative user's answer.

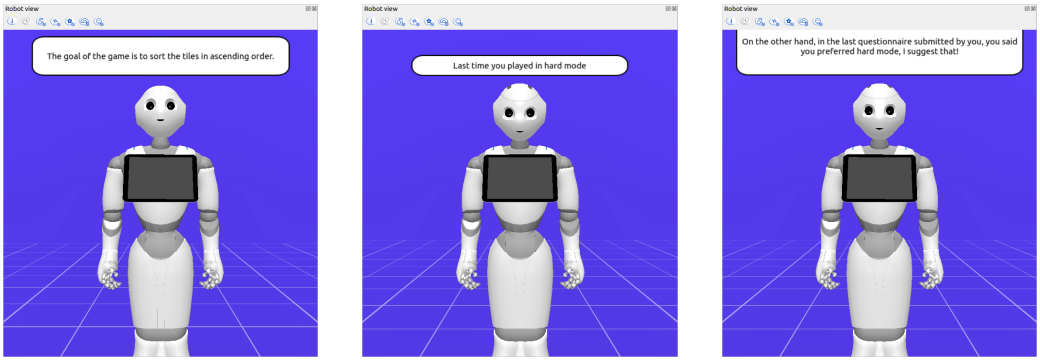
```
Pepper says:  Do you want to play with me? ['yes', 'no']
Human answer: yes
```

Figure 15: Tablet interaction for "yes" or "no".

Pepper makes a difference between the case of a new and old user when the answer is *"yes"*:

- If the user has never been met, Pepper starts a little questionnaire where it asks whether the user knows about the Slide puzzle game, and if it receives a positive response, the page explained in Section 4.3.2 opens while Pepper makes an introduction to the game explaining the rules and the goal of the game (Figure 16a). After having completed the tutorial successfully, the user is directed to the page of the *easy* level (being an user that has never played the game) and can start the game. If the user has given a negative response to this question (so he/her knows the rules and so on), the game page of the difficulty selection is opened and the game starts.
- If Pepper already knows the user, pepper gives information about the last difficulty played and what is the preferred one (reading the submission form from the survey). The user is free to choose the level he wants, Pepper does only make suggestions.

In the case in which user's answer is *"no"*, Pepper says goodbye to the user making an animation.



(a) Pepper explains the rules. (b) Last difficulty played. (c) Favourite difficulty.

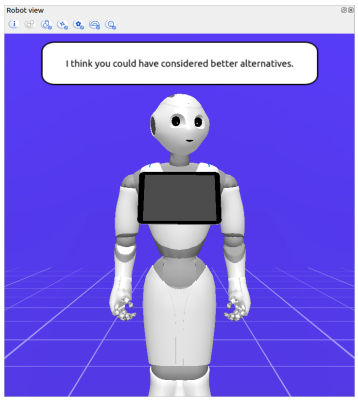
Figure 16: Pepper's interaction with an user.

When the starting page of the game starts, the user is able to perform four moves and Pepper two. Pepper is involved during the game and gives advices (with random sentences) to the user regarding the difference between user's moves and what the robot plans at each iteration:

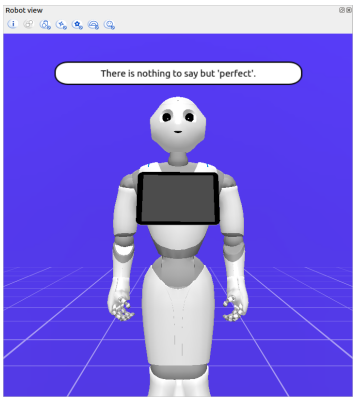
- When Pepper thinks that the user has not performed the best moves he could make (compared to the optimal plan computed) to achieve the solution, it points it out as shown in Figure 17a. In this case the robot does not eventually delete the user's moves, but it adapts to the situation and makes its moves accordingly.
- In the case in which Pepper agrees with the user's moves, it congratulates him/her (Figure 17b).

- If Pepper has to think about the next move because the steps made by the user are not the best, but good, Pepper performs the thinking pose (Figure 17c).

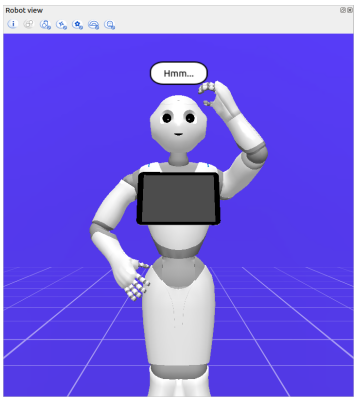
This in an iterative process in which Pepper computes the next move at each step and it specifies the direction he chooses with a move that simulates the slide of a tile on the board (as shown in Figure 17d).



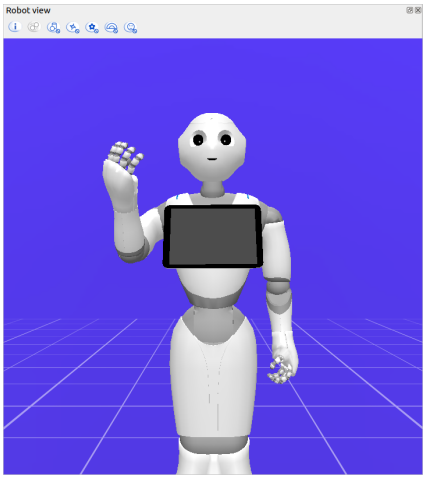
(a) User has made bad moves.



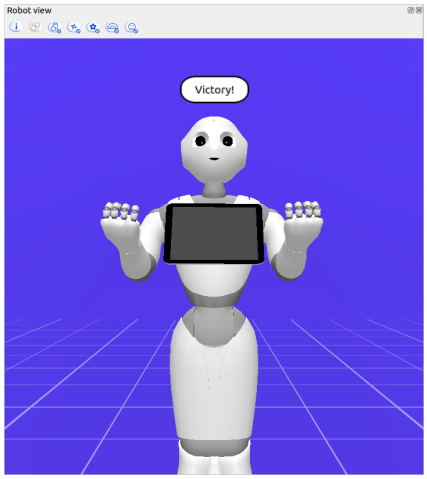
(b) User has made good moves.



(c) Thinking about the move.



(d) Move Up.



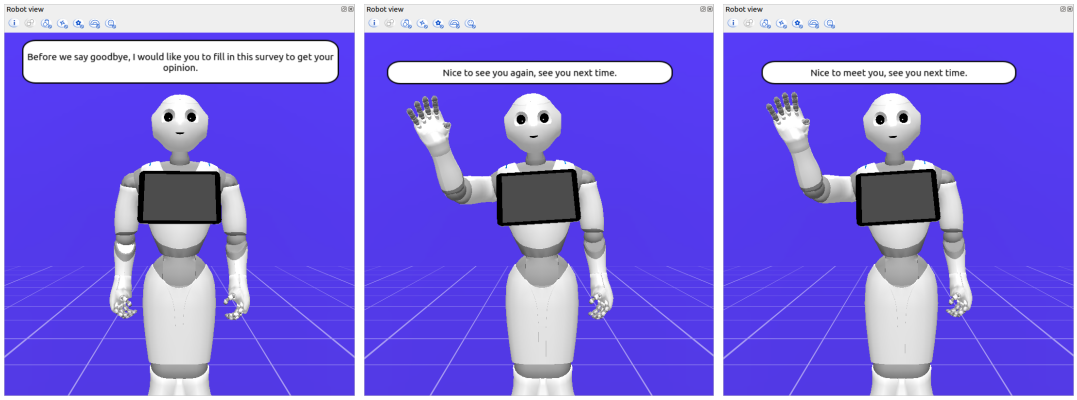
(e) Victory animation.

Figure 17: Pepper's animations during and after the game.

When the game ends because all the tiles are sorted, Pepper performs a victory animation and congratulates with the user (Figure 17e).

5.4 Final Interaction

After having completed the game, Pepper asks if the user wants to make a new game. If the user wants to play again, the web page that allows the selection of the difficulty opens with the wins counter updated. If this is not the case, Pepper asks the user to compile the survey explained in Section 4.3.4 that will be displayed in order to give his/her opinion about the interaction with Pepper (Figure 18a). The project ends when the survey is submitted (and stored for future interactions as already mentioned) and Pepper says goodbye either to an old or new user telling different sentences for the two cases (Figure 18b and 18c).



(a) Pepper asks to complete the survey.

(b) Pepper says goodbye to an old user.

(c) Pepper says goodbye to a new user.

Figure 18: Pepper different scenarios for saying goodbye.

6 Conclusions

In today's world, the proliferation of robots designed to interact with humans is becoming increasingly prevalent. These interactions can range from simple entertainment for the elderly to educational tools for children. This project was very interesting because we successfully developed an interface that aimed to be as intuitive as possible, allowing users to perceive themselves as collaborating with the robot, which was a gratifying achievement. The most intricate aspect of the entire project was programming the robot's behavior to be as 'human-like' as possible, ensuring genuine interaction with the user rather than merely serving as an accessory. It was a challenging task, but we are pleased with the final outcome. Furthermore, our work holds the potential for further extensions in both HRI and RA.

- Future directions may involve testing our work on a physical robot, leveraging all available sensors to evaluate the sensing-related software component.
- We could allow Pepper to take a picture of the user and use it as puzzle image, enhancing the user experience.
- Consideration might be given to implement a Multi-Agent planner to further enhance performance and introduce more complex and challenging levels for the user.
- Exploring the use of an offline non-deterministic planner could be beneficial, as currently, the plan is generated at the time of level creation.

In conclusion, this project not only gave us a chance to face the potential of social robots in performing collaborative activities with humans, but also demonstrated the feasibility of robots interacting with humans in a social context. The use of Pepper for this collaborative game illustrated the possibility of using robots to test particularly difficult games that a human may struggle to finish alone, and this can have wider applications to collaboration of robots in general when performing daily life tasks. A robot help can always be useful if well performed and trustable.

References

[1] Phoebe Liu, Dylan F. Glas, Takayuki Kanda, Hiroshi Ishiguro, and Norihiro Hagita. How to train your robot - teaching service robots to reproduce human social behavior. In *The 23rd IEEE International Symposium on Robot and Human Interactive Communication*, pages 961–968, 2014.

[2] Phoebe Liu, Dylan F. Glas, Takayuki Kanda, and Hiroshi Ishiguro. Data-driven hri: Learning social behaviors by example from human–human interaction. *IEEE Transactions on Robotics*, 32(4):988–1008, 2016.

[3] Takayuki Kanda, Takayuki Hirano, Daniel Eaton, and Hiroshi Ishiguro. Interactive robots as social partners and peer tutors for children: A field trial. *Human–Computer Interaction*, 19(1-2):61–84, 2004.

[4] Malcolm Doering, Dylan F. Glas, and Hiroshi Ishiguro. Modeling interaction structure for robot imitation learning of human social behavior. *IEEE Transactions on Human-Machine Systems*, 49(3):219–231, 2019.

[5] Filipa Correia, Samuel Mascarenhas, Rui Prada, Francisco S. Melo, and Ana Paiva. Group-based emotions in teams of humans and robots. In *Proceedings of the 2018 ACM/IEEE International Conference on Human-Robot Interaction*, HRI '18, page 261–269, New York, NY, USA, 2018. Association for Computing Machinery.

[6] Gabriel Skantze, Martin Johansson, and Jonas Beskow. Exploring turn-taking cues in multi-party human-robot discussions about objects. In *Proceedings of the 2015 ACM on International Conference on Multimodal Interaction*, ICMI '15, page 67–74, New York, NY, USA, 2015. Association for Computing Machinery.

[7] Sarah Strohkorb Sebo, Priyanka Krishnamurthi, and Brian Scassellati. "i don't believe you": Investigating the effects of robot trust violation and repair. In *2019 14th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 57–65, 2019.

[8] Séverin Lemaignan, Mathieu Warnier, E. Akin Sisbot, Aurélie Clodic, and Rachid Alami. Artificial cognition for social human–robot interaction: An implementation. *Artificial Intelligence*, 247:45–69, 2017. Special Issue on AI and Robotics.

[9] Rachid Alami, Mathieu Warnier, Julien Guitton, Séverin Lemaignan, and Emrah Sisbot. When the robot considers the human... 01 2011.

[10] Daniel Foead, Alifio Ghifari, Marchel Budi Kusuma, Novita Hanafiah, and Eric Gunawan. A systematic literature review of a* pathfinding. *Procedia Computer Science*, 179:507–514, 2021. 5th International Conference on Computer Science and Computational Intelligence 2020.

[11] Neng-Fa Zhou and Hakan Kjellerstrand. Solving several planning problems with picat. In *The 26th Chinese Control and Decision Conference (2014 CCDC)*, pages 346–350, 2014.