Elective In Artificial Intelligence - HRI and RA
RushHour: A Collaborative Social Robot

Alessio Sfregola 1798423
Caterina Borzillo 1808187
Federica Cocci 1802435

March 15th, 2023

SAPIENZA
UNIVERSITÀ DI ROMA

All authors have contributed to the project in an equal way.

Contents

1 Abstract

RushHour is a social and interactive robot for people's entertainment. He plays with the human the Rush Hour game in a collaborative way: human and robot work together to reach the goal making one move alternately. The social interaction consists in an introductory presentation, in an initial and final surveys and in different robot animations also during the game. The robot reasoning aspect is developed through AIPlan4EU framework. The physical communication between the several components of the project has been developed using a system of clients and servers.

2 Introduction

Artificial Intelligence (AI) and social robots are rapidly transforming the way we interact with technology and with each other. Social robots are machines designed to simulate human behavior and communication, often through the use of AI. These robots can take many forms, such as humanoid robots, pet robots, or virtual assistants, and they are increasingly being used in a variety of settings, including homes, schools, hospitals, and public spaces. The potential benefits of social robots are numerous. They can provide companionship for the elderly, assist individuals with disabilities, and support children's education and development. They can also perform tasks that are dangerous or repetitive for humans, such as cleaning or manufacturing. Furthermore, they can be used to collect data and perform analysis in areas such as healthcare and transportation, leading to better decision-making and outcomes. In our case, we want that our social robot, in addition to experiencing the social field with the human, is also equipped with intellectual abilities that enable it to reason and solve the game along with the human. The way the overall interaction is established is very important regarding the acceptability aspect of the human towards the robot. The rise of social robots has to deal, in fact, also with this aspect: how to make the robot behave toward the human so that the human has the best possible experience. Overall, the integration of AI and social robots in society has the potential to revolutionize our lives in positive ways, but it is important to carefully consider the implications and ensure that these technologies are developed and used responsibly.

The basis idea of our project is to develop a robot, called RushHour, that solve the Rush Hour game cooperating with the human. Rush Hour 1a is a sliding puzzle game whose objective is to move a red car through a grid of other cars and trucks to the exit of the game board. Here are the basic rules of the game: the game board consists of a 6x6 grid of squares where cars (length 2) and trucks (length 3) are placed vertically or horizontally; the player moves the cars and trucks by sliding them along the grid horizontally or vertically, but not diagonally in order to clear a path for the red car to reach the exit, which is located on the edge of the grid opposite the starting position. In our version of Rush Hour, a move corresponds to a sequence of movements all in the same direction relative to the same vehicle: man and robot take turns at the board each time a different vehicle moves or the direction of the movement of the same vehicle changes. These rules brought us to develop the reasoning ability of the robot in order to let him calculate the best next move allowing it to win the match in the fastest way possible, taking into account the configuration of the board and its changes once the human makes a move.

The reasoning aspect of our project has been implemented using a Python framework called AIPlan4EU [1] where, in order to define a planning problem, we have to declare different types of objects, fluents, actions with their preconditions and effects, initial state and goal.

Concerning human-robot interaction (HRI), we want to get the interaction between human and robot as natural as possible. The idea is to use our software on a physical robot called Pepper [2] which is from the NAOqi robots family [3] 1b. For this purpose, NAOqi has been used as it provides APIs, for example for speech recognition and synthesis capabilities, that allows the robot to understand and respond to spoken commands and to communicate with users through speech. It also includes a range of sensors, such as cameras and microphones, that enable Pepper to perceive and interact with its environment. One key aspect

of NAOqi that makes it well-suited for human-robot interaction is its modular architecture. The platform is designed to be easily extensible, allowing developers to add new behaviors and capabilities to the robot as needed. Actually we have tested our software only on a simulated Pepper 1c. In the simulated environment the robot can act as if he was physical. We use Choregraphe Suite [4] as platform for the simulation. In Choregraphe some of the tools of the physical NAOqi are not enable such as sonars or verbal speaking but we have find some expediences to test however our code for the sensing part. The way the robot is represented in its naturalness is crucial for the success of the human's experience with the robot. The fact that the robot, after having introduced itself with the human, learns his name and continues to interact with him using the user's name, gives a sense of familiarity and welcome in the human's mind and this is fundamental in those situations in which, for example, there is a robot that has to deal with children (educational field) or elderly people that have in some way start a process of "acceptability" of the robot in order to interact with him in the correct way.

Thanks to JavaScript (JS) and CSS and HTML, we created a web application to reproduce the Pepper tablet. This web application allows the user and the robot play together since the human can visualize the progress of the game and play the next move for solving the problem. Finally, in order to make all these components communicate with each other, we have exploited websockets and in order to implement them in Python we have used Tornado [5].

(a) Rush Hour (b) NAOqi family (c) Virtual Pepper in Choregraphe

Figure 1: Rush Hour and Robots

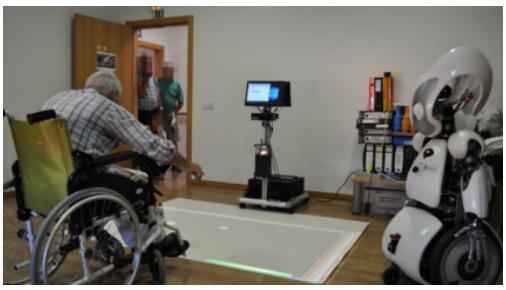
3 Related Works

Social robots that play games are an emerging field of robotics that is rapidly gaining popularity. These robots are designed to interact with humans in a social setting and engage in various games with them, such as board games, card games, and video games 2. The primary goal of these robots is to provide entertainment and companionship to humans, while also promoting social interaction and cognitive development. With advances in artificial intelligence and robotics, social robots are becoming more sophisticated and capable of adapting to individual human behavior. This has led to an increasing interest in the potential applications of social robots in areas such as education, therapy, and elderly care. Overall, social robots that play games are a fascinating and exciting development in the field of robotics, with the potential to transform the way we interact with technology and each other.

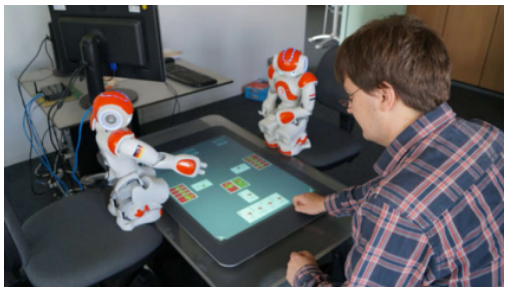
3.1 Human-Robot Interaction

Among the researchers that for decades are making great contributions to HRI field there is certainly the prominent researcher Rachid Alami, head of the Robotics and Artificial Intelligence group at LAAS-CNRS in Toulouse, France. Alami research field concerns the way robots can work and interact together with humans in real-world environments. His studies focus on one side on the development of intelligent robots that are capable of performing complex tasks in unstructured environments, such as homes, hospitals, and factories and on the other side on the development of robots that can understand and respond to human gestures and speech, making them more natural and intuitive and that for example can be used in fields such as human-robot collaboration, assistive robotics, and social robotics. For example, in the paper [6] Alami and other scientists discuss techniques for human and robot interaction in which themes as joint action, adaptation, and entrainment are highlighted: an important challenge, they say, could be developing algorithms and architectures that can enable robots to perceive, reason, and act in a way that is compatible with human behavior. These aspects could be related to our project in terms of finding that specific human-robot sequence of moves that solves the Rush Hour game in the shortest time and in the least number of moves or related in terms of implementing a deeper collaboration format in which the robot helps the human to reason about the game and the best next move.

Another topic explored during the course concerns the psychological impact that a fragile robot produces on humans; in particular, the paper [7] states that the concept of fragility in robots can cause a sort of affection and familiarity in humans' minds such that the (human-robot) relationship established becomes more powerful. This fragility aspect inducted in robots could be applied for example in our project by considering the robot not as a perfect intelligent robot but as someone that has to learn some knowledge from the human: this way of reasoning could create in the user the need to take care of the robot by expressing a more active behavior toward the robot for the purpose of enhancing the social relationship with him.



(a) Elderly Care Facility



(b) One Human and Multiple Robots playing

Figure 2: Interactions human-robot

3.2 Reasoning Agents

Rush Hour has been the subject of various studies in the field of artificial intelligence, particularly in the area of planning. It has been a popular test-bed for artificial intelligence research since its inception in the late 1990s since the game presents an interesting challenge for AI systems due to its complex and dynamic nature, which requires the system to reason about how to move vehicles on a crowded grid to reach a target location. Over the years, researchers have applied a variety of AI techniques to solve the Rush Hour puzzle, including search algorithms, constraint satisfaction, planning, and even neural networks. Researchers have developed a variety of planning algorithms for the Rush Hour game, including real-time heuristic search, PDDL-based planning, and constraint satisfaction algorithms. In [8] the authors provide a

constraint programming encoding in MiniZinc and a model in Answer Set Programming for the game. In [9] they evolve heuristics to guide IDA* search for the 6x6 and 8x8 versions of the Rush Hour puzzle: starting from several novel heuristic measures the authors turn to genetic programming (GP).

In our case, we decide to use AIPlan4EU (`https://www.aiplan4eu-project.eu/`) which allows us to access existing planning technologies.

4 Solution

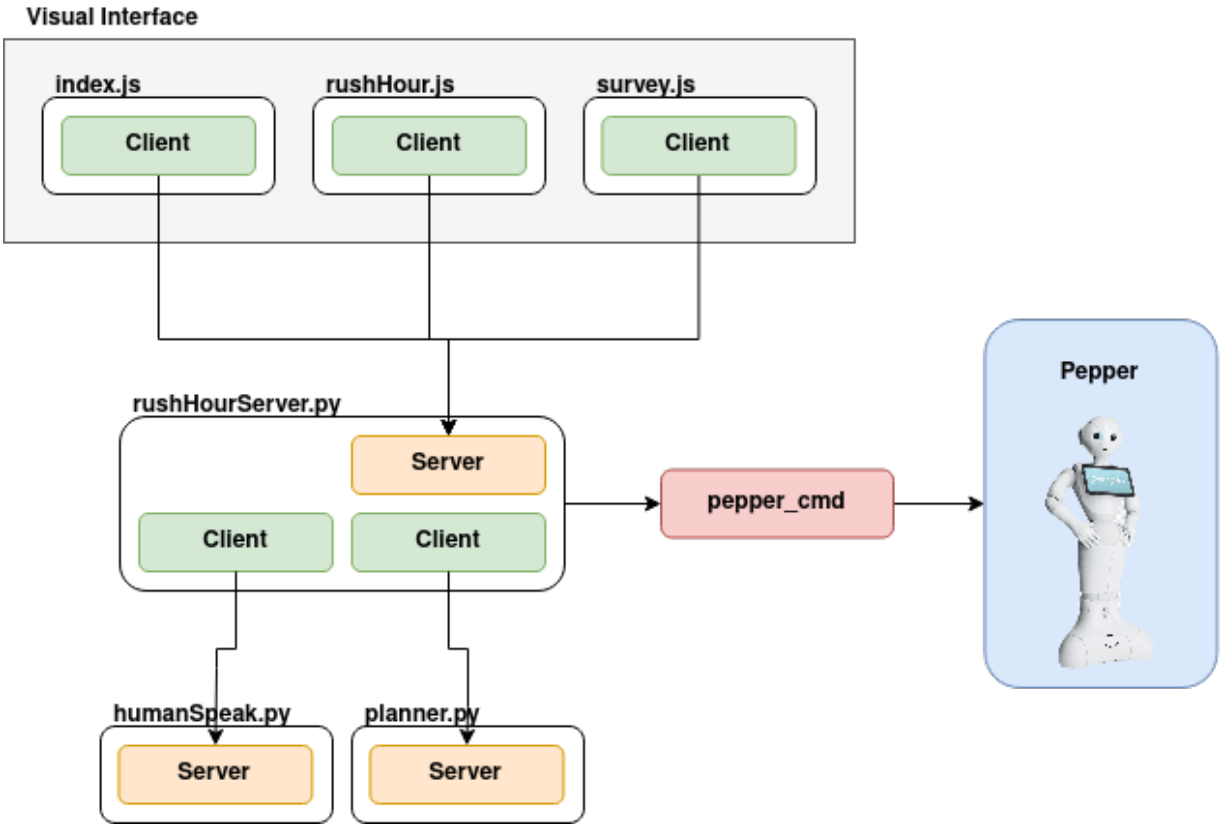


Figure 3: Proposed architecture.

The architecture of the proposed solution is displayed in 3. The core of the structure is the main server (*rushHourServer.py*), which is used as a crossroads that puts into communication all the different modules of the project and it is written in Python. We chose to put the rushHourServer as a crossing point for every communication because the functions contained in *pepper_cmd* are only accessible there. So each time a message is received by the server, we can trigger a Pepper behavior (motion or speech) relative to the current event. The flow of information follows the following scheme:

1. When the interface is set up, a request to determine the state of the game is sent to the main server. We could be in two situations: the interface has been loaded for the first time since a user completed the interaction, or it has been loaded after one instance of the game has been solved. Depending on which situation we are in, the display shows different information.

2. If the interface loads for the first time, it stays in an idle state until the server sends a message indicating that a human has entered Pepper's area of interaction (less than 1.5m in front of the robot). In this case the interface loads the initial survey and shows it to the user.

3. Each text shown in the interface is sent to the main server, which triggers an event that results in Pepper saying said text. This way, the user can read or hear the question asked by Pepper.

4. During this survey, the *humanSpeak* module comes into play. In fact, given has two means to understand the request made by Pepper (hearing or reading), we gave them the ability to interact in both ways, by speaking or pressing buttons on the tablet. Given that we are not using the real robot, but a simulated environment, it's not possible to actually speak to the robot, but to simulate this, the *humanSpeak* module features a console that prompts the user to write what they would like to say to the robot. The user is able to choose between a fixed set of words, that the robot is programmed to understand, which correspond to the text visible on the buttons they are able to press on the interface. This data relative to the current question is sent to a server hosted in the module, which displays it in the console, and waits for a response by the user. They have 10 seconds to say something. If they don´t say anything, we assume they want to interact with the robot via the tablet. If they do say something (i.e. write it in the console), the message is sent to the main server, which forwards it to the interface, prompting a change of scene. The behavior is exactly the same as that obtained when pressing the corresponding button. The mean of communication with Pepper can be chosen freely at each moment, independently from what the user chose in previous instances.

5. At the end of the survey, the interface sends the temporary result to the main server, which stores them in the main server, waiting for the additional results produced in the final survey after the match. The game interface is loaded.

6. As the game interface appears, a request for the data relative to the chosen level is sent to the server, which responds with the desired configuration of the board. At this moment, the user can start playing.

7. After each move, a message is sent to the main server, containing the updated configuration of the board. The server forwards this information to the other server module in our architecture, the *planner*. It receives the data about the board configuration from the main server, and computes the desired plan for the solution of the game. From this plan, we extract just the first move, and we return it to the visual interface, which updates the board accordingly. After this move performed by Pepper (to which a coherent animation of the robot corresponds), the turn passes again to the user, and the cycle repeats.

8. The game ends when Pepper, or the user, make a move that brings the red car in the goal position. At this point the interface changes again, showing a screen where the user is asked whether they want to play another match, or start a survey regarding their experience.

9. In the first case the user is brought to the choosing level screen, while in the other scenario the interface changes, showing the survey.

10. When the survey loads, a request is made to the main server, which responds with the object containing answers the same user gave during the initial survey, which is then updated with the answers given in this final survey, and sent again to the server.

11. This marks the end of the interaction with the robot, which goes to the initial state, waiting for a human to approach it.

The diagram visualizing this interaction flow is shown in 4, and described in more detail below, with less focus on the technical implementation, and more details regarding the Human-Robot Interaction.

4.1 Human-Robot Interaction

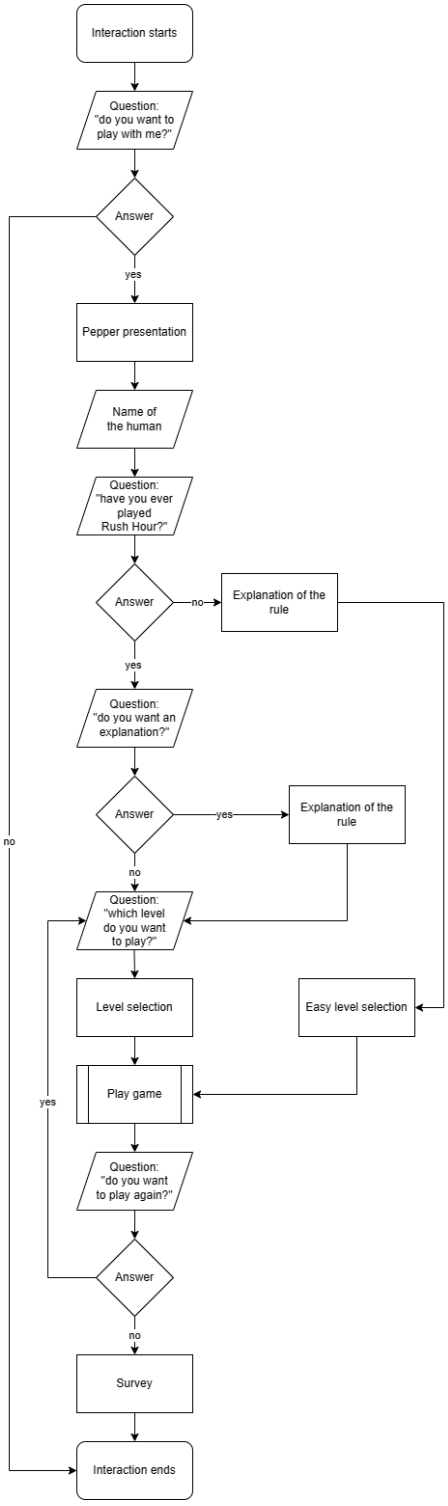
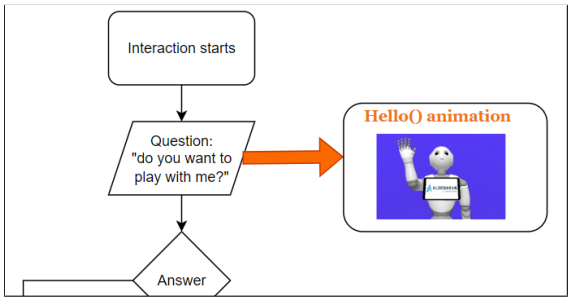


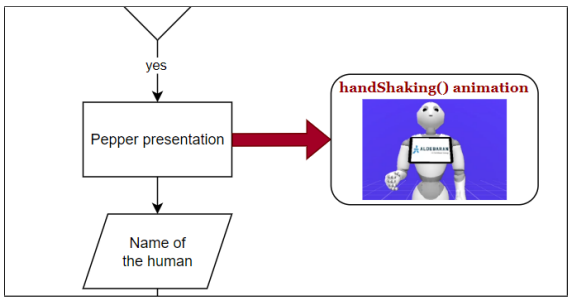
Figure 4: Flow diagram of the HRI part.

The flow diagram concerning Human-Pepper interaction is shown in Figure 4. Initially Pepper waits in an idle state for a human to approach it. Its display shows the text "Waiting for someone to play with me...", inviting people to come closer to it. Once a person gets to a distance less than 1.5 meters, Pepper detects them with its frontal sonar and starts the interaction. To be clear and immediately state its intentions, Pepper asks the human whether they want to play a match of Rush Hour game with him; the human may accept, or refuse. In the latter case, Pepper communicates its displeasure to the user and after some seconds re-display the initial page, waiting for another user to connect with. If the user accepts, on the other hand, Pepper starts a small survey, which will be stored in the server, regarding the user and past experiences with the game. It will prompt the user for a username, which will be used to store their data, and whether they have already played the game, or need to be reminded the rules. If from this information, the fact that the user doesn't have experience with the game emerges, then Pepper automatically chooses to let the user play the easiest level available. Otherwise, the choice of the difficulty level is left to the human agent.
After this small survey, the second phase starts, with the human and the robot playing the game in a collaborative fashion. During the game, the two players in turn make their move until the goal is reached, at which point the game ends.
In the third and final phase, Pepper asks whether the user wants to play again or whether it wants to end the interaction, inviting the person to fill another short survey regarding their experience with the game and the robot. The answers to these questions will be joined with those obtained in the first phase and stored in the server. On the other hand, if the user is up to play another match, the interface leads them to the level choice screen, in which the user is able to choose which level they want to play next. To help the user choose wisely, Pepper suggests to play a difficulty level slightly higher than that just played by the user, hoping to make the user face a bigger and more compelling challenge.

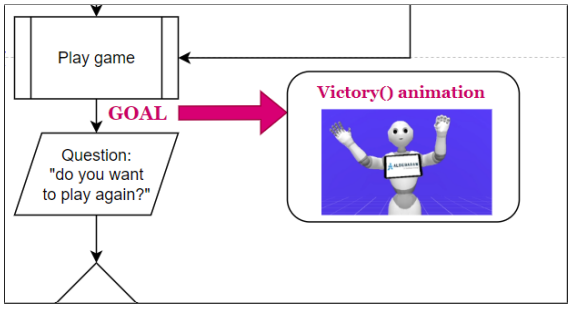
Concerning the Pepper gestural communication, in the images below it's possible to notice when and how four different animations are realized.



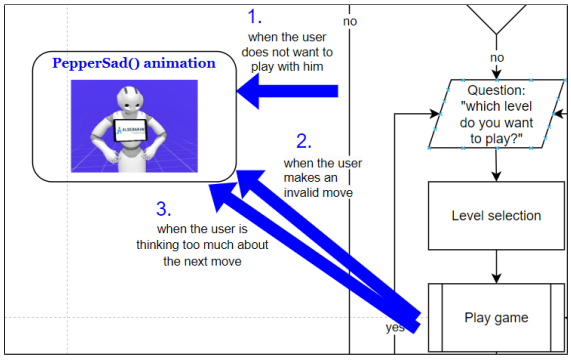
(a) When and how Pepper hello animation occurs.



(b) When and how Pepper hand shaking animation occurs.



(c) When and how Pepper victory animation occurs.



(d) When and how Pepper sad animation occurs.

Figure 5: Examples of four different Pepper animations.

Aside from these animations represented in Figure 5, we created four additional animations that the Robot execute whenever he makes a move in a certain direction (for details see in the next section).
The reason why the robot gestural communication is fundamental is that it enhances the acceptability aspect of the human toward the robot. In fact, we tried to make the robot's moves as natural as possible in order to improve in the human the sense of confidence and familiarity against the robot, thing that is very important in a lot of social contexts like the medical and the educational one, when the robot has to deal with children and elderly people.

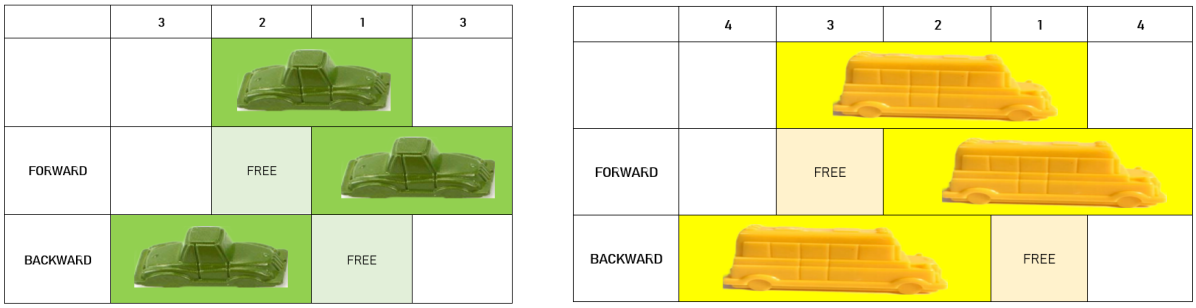
4.2 Reasoning Agents

Concerning the reasoning part, as we have already told, in Rush Hour vehicles can only move vertical or horizontal and in particular each vehicle can move forward or backward. In order to implement all of this, first of all we have set a convention of how the vehicle configuration is passed: the first cell is always the "head" of the machine. If the vehicle is towards right, the forward will consist in moving towards right and the backward will consist in moving to left; if the vehicle is towards up, the forward will consist in moving towards up and the backward will consist in moving towards down (see figures 6 and 7) [for vehicles looking left/down the logic is specular].

The red car is always instantiated as an horizontal car looking to right and so the goal of the planning problem is reached when its head is at the rightmost cell of the same row.

To set the initial configuration of the planning problem, we will pass two arrays: the former contains all the cars and their position on the board (the red car is always the first one of this list), the latter is the list of all the trucks and their positions. For each car we need ***(id, cell1_row, cell1_col, cell2_row, cell2_col)***; for each truck we will need ***(id, cell1_row, cell1_col, cell2_row, cell2_col, cell3_row, cell3_col)***. The id is a progressive number representing the vehicle in the board and in this way we always know which car/truck moves.

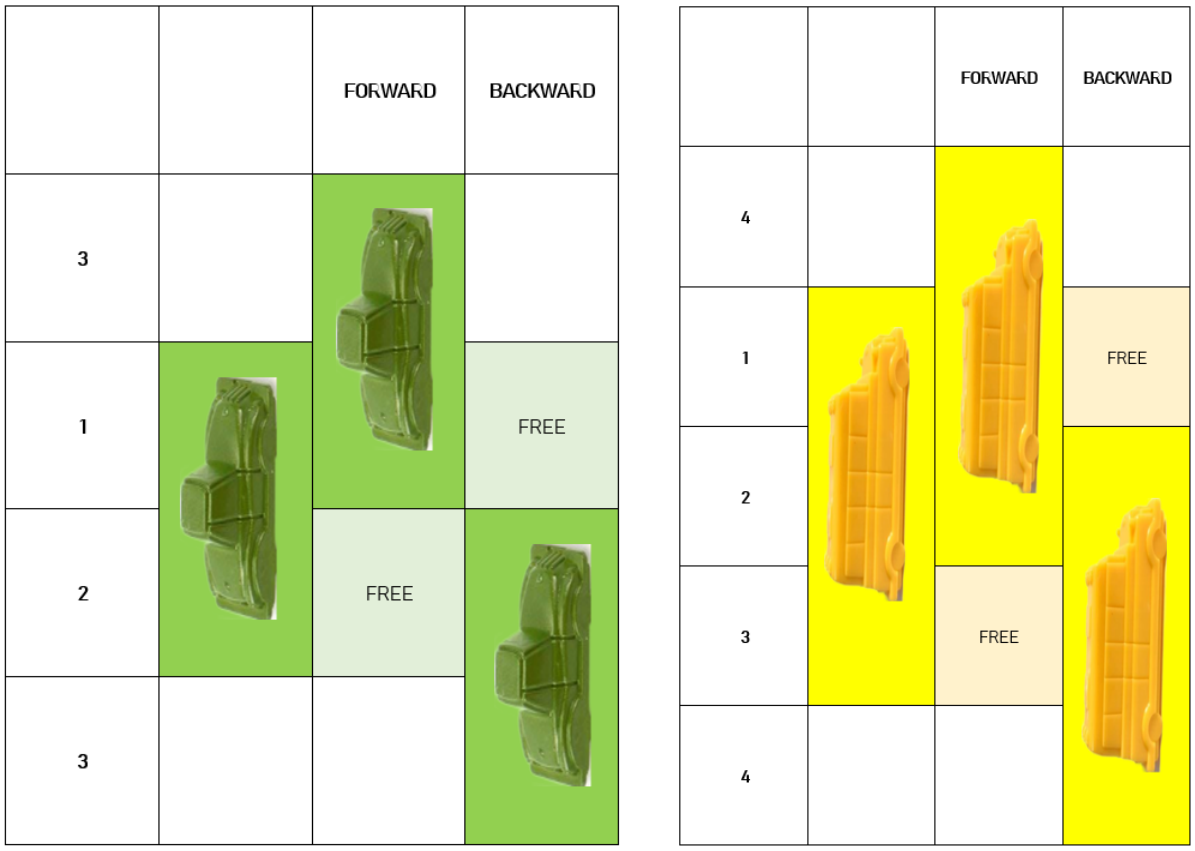
As previously said, we resolve the reasoning aspect using AIPlan4EU and in particular we will return a sequence of actions where each action is one-cell movement of a vehicle. We will give to the HTML the id of the vehicle, the motion type i.e. forward or backward and how many cells the vehicle moves.



(a) Moves for horizontal cars

(b) Moves for horizontal trucks

Figure 6: Moves for horizontal vehicles



(a) Moves for vertical cars

(b) Moves for vertical trucks

Figure 7: Moves for vertical vehicles

5 Implementation

This section provides a detailed explanation of how each component of the project has been implemented, from the tools and frameworks that have been used to the reasons motivating our choices.

5.1 Experimental setup

The project has been developed fully in a Linux environment. For the installation of the tools used to communicate with the robot Pepper we used a docker image with all the packages and programs necessary to do so.

Given that we didn't have the physical robot at our disposal, we had to rely on a simulated version of Pepper. We opted to simulate it in Choregraphe, an application developed by Aldebaran Robotics, which is very useful to design and develop animations, behaviors and dialog for robots, and test them in simulated environments.

To write the functions of the modules that compose our architecture (see Figure 3), we decided to use

Javascript, HTML and CSS for the components that make up the visual interface displayed in Pepper's tablet, while we used Python for the programs that manage the definition and solution of the planning problem, and for the development of the main server, that manages all the communications between the different components of the project.

To exchange data between these components, we made use of WebSockets. Javascript natively supports these means of communication, so we didn't have to use any external library, while for Python programs, we decided to use Tornado, a framework used to create and manage scalable web servers in Python.

5.2 Human-Robot Interactions

5.2.1 Use of *Pepper_tools* modules for Pepper communication

The process that sees Pepper involved in the interaction with the human happens through a python program in which we make use of *pepper_tools* utilities. *pepper_tools* is a repository that contains different tools to manage the Softbank Pepper robot. As we can see from image shown in figure 8, there are multiple modules in pepper_tools that make Pepper perform different tasks: using its sensors, making some physical moves, managing audio inputs and outputs or saying something.

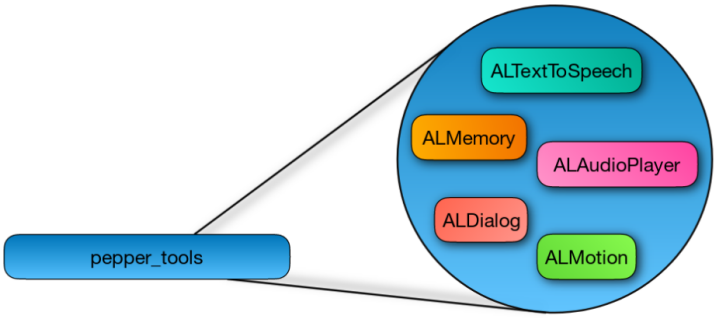


Figure 8: Modules in *pepper_tools*.

First of all, once we establish the connection with the simulated robot, we make use of some *pepper_tools* functions to read the information provided by the virtual sonar sensor. Whenever the distance of the user from the robot is less than a meter and a half, the interaction begins. Actually, since we work with simulated Pepper we made sure that, whenever we say that we "enter scene", the distance from the robot is 1,20 meters: from this moment the interaction starts.

```
# Pretend to scan the environment for a human. In reality, the human position is passed as a parameter to the function
# The position is then inserted into the memory relative to the sonar sensor and retrieved as we would do in a real scenario
# The function returns True if a human closer than 1.5m is detected, False otherwise
def scanEnvironment(robot, humanPos):

    robot.memory_service.insertData(memkey['SonarFront'], float(humanPos))

    # Detect human only if at a distance of less than 1.5 meters
    humanDetected = True if robot.memory_service.getData(memkey['SonarFront']) < 1.5 else False
    print 'SonarFront: ' + str(robot.memory_service.getData(memkey['SonarFront']))

```

```
    return humanDetected
```

Then, in order to make the robot speak, the ALTextToSpeech module is used. In particular, ALTextToSpeech sends the sentence that the robot has to say to a text-to-speech engine and the result of this synthesis is sent to the robot. The module also endorses voice customization, such as speed or language. This module is used through the call of the function *say* that is utilized whenever, during the interaction, the robot starts saying something to the human. Actually, since we use the simulated robot, we will know that Pepper on Choregraphe is saying something by seeing a dialogue box above him. In the image below, an example of how the function is called is shown.

```
        elif ('wonGame' in message):
            robot.say('Congratulations! We solved the puzzle!')
            pepperVictory(robot)

        elif ('pepperSad' in message):
            config = json.loads(message)
            robot.say(config['sentence'])
            pepperSad(robot)
```

Another use of *pepper_tools* modules that makes the robot move is the ALMotion module. We developed 8 python customized functions that make the robot do some animations during the interaction, starting from the the initials "Hello" and "Hand Shaking" motions that the robot make to greet the human, to the movements that the robot make to shift a car or a truck during its move in the game, to the final Pepper reaction if the game ends in a victory. All the functions are manually implemented and each of these present different timelines. Above it's showed the "Hand Shaking" function that is executed by the robot in the initial presentation phase, after the "Hello" motion, in order to introduce itself to the human.

```
def handShaking(robot):
  # animazione del robot quando stringe la mano all'umano
  session = robot.session_service("ALMotion")

  isAbsolute = True

  # pepper raises his arm and opens his hand
  jointNames = ["RElbowRoll","RElbowYaw","RHand","HeadPitch","HeadYaw"]
  jointValues = [1.25,1.63,0.98,0.052,-0.31] #in degree [71.61,93.39,0.98,3,-18]
  session.angleInterpolation(jointNames, jointValues, 0.8, isAbsolute)
  # pepper shakes the hand of the human
  for i in range(3):
    jointNames = ["RElbowRoll","RHand"]
    jointValues = [1.53,0.54] #in degree [87.66,0.54]
    session.angleInterpolation(jointNames, jointValues, 0.3, isAbsolute)
    jointValues = [1.15,0.54] #in degree [65.89,0.54]
    session.angleInterpolation(jointNames, jointValues, 0.3, isAbsolute)
    if i==2:
      jointNames = ["RElbowRoll","RHand"]
      jointValues = [1.53,0.54] #in degree [87.66,0.54]
      session.angleInterpolation(jointNames, jointValues, 0.3, isAbsolute)
  # pepper opens his hand to release the human hand
  session.angleInterpolation("RHand", 0.98, 0.3, isAbsolute)

  robot.normalPosture()
  return
```

Other examples of animation are the four possible moves that the robot can make during the game: moving a car or truck horizontally on the right or on the left or moving a car or truck vertically up or down; the code of the "MoveVerticalUp" animation is specified above in figure *.

```
def moveVerticalUp(robot):
  # animazione del robot quando muove la macchina in verticale verso l'alto
```

```
  session = robot.session_service("ALMotion")
  isAbsolute = True

  # move posture
  jointNames = ["RElbowRoll", "RElbowYaw", "RHand", "RShoulderPitch", "RShoulderRoll", "
    RWristYaw"]
  jointValues = [1.34, 1.39, 0.92, 1.20, -0.26, -1.17]
  times = [0.8, 0.8, 0.8, 0.8, 0.8, 0.8]
  session.angleInterpolation(jointNames, jointValues, times, isAbsolute)

  # hand grasps cars
  session.angleInterpolation("RHand", 0.52, 0.3, isAbsolute)

  # arm that slides to move the car
  jointNames = ["RElbowRoll", "RShoulderPitch"]
  jointValues = [0.91, 0.83]
  times = [0.6, 0.6]
  session.angleInterpolation(jointNames, jointValues, times, isAbsolute)

  robot.normalPosture()
  return
```

Finally, we implemented two additional animations: the first, "PepperVictory", is called when the human and Pepper win the game and the second, "PepperSad", is called (in three different situations) when the user decides to not play the game with him, when the user is thinking too much about the next move to make and when the user makes an invalid move.

```
def pepperSad(robot):
  # animazione del robot quando perde
  session = robot.session_service("ALMotion")

  isAbsolute = True
  # defeat posture
  jointNames = ["RElbowRoll", "RElbowYaw", "RShoulderPitch", "RShoulderRoll", "LElbowRoll",
    "LElbowYaw", "LShoulderPitch", "LShoulderRoll", "HeadPitch", "HipPitch"]
  # jointValues_gradi = [79.2, 21.9, 76.3, -40.0, -81.6, -19.1, 71.1, 40.1, 25.5, -30]
  # jointValues is expressed in radiant, in particular: [rad/10, rad/10, rad/10, rad, rad
    /10, rad/10, rad/10, rad/10, rad/10, rad]
  jointValues = [1.38, 0.382, 1.33, -0.69, -1.42, -0.333, 1.24, 0.699, 0.445, -0.52]
  times = [0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6]
  session.angleInterpolation(jointNames, jointValues, times, isAbsolute)
  [22.0, -22.0]

  # pepper shakes his head
  # gradi = 22.0 --> rad = 0.38
  session.angleInterpolation("HeadYaw", 0.30, 0.4, isAbsolute)

  # gradi = -22.0 --> rad = -0.38
  session.angleInterpolation("HeadYaw", -0.30, 0.4, isAbsolute)

  # gradi = 22.0 --> rad = 0.38
  session.angleInterpolation("HeadYaw", 0.30, 0.4, isAbsolute)

  robot.normalPosture()
  return
```

As we will better explain in the following, the robot suggests a level to the user by performing some simple reasoning over the received answers. After the questionnaire, the user can select the level he wants and start playing. In order to collaboratively solve the game, the user and the robot need to execute a move alternately. While doing so, the robot informs the user when it is its turn to play and mimics to the user the direction in which Pepper has decided to move the object. This movement is implemented through a custom function similar to *doHello()*. While playing, Pepper also warns the user when he tries to make an

invalid move, such as picking from an empty rod or trying to move a disk on a smaller one. This is done by receiving messages from the web application through a websocket, as will be better explained in one of the following sections. The warning is given to the user by using the aforementioned *asay*, whose name stands for animated say, meaning that the robot gesticulates while speaking.

```
            elif(received[0]=="Victory"):
                print("%s!!Victory!!%s" %(GREEN,RESET))
                pepper_cmd.robot.say('Victory')
                victoryDance();
            elif(received[0]=="moveToLeft"):
                doMoveToLeft()
            elif(received[0]=="moveToRight"):
                doMoveToRight()
            else:
                print("%s!!uNKNOWN ERROR!!%s" %(RED,RESET))
```

Finally, when the robot and the user manage to win by moving all the disks to the rod on the right, the robot informs the user while making a little victory dance, that has been implemented similarly to the *doHello* function, leveraging the ALMotion module. The robot then invites the user to rate his experience and the difficulty he encountered using the tablet, as we will better explain later, and he can then play again or leave.

5.2.2 Website for interaction through the tablet

During its period of activity, Pepper keeps showing a visual interface on its tablet. This interface is developed using the standard technologies that are commonly used when developing applications that run on a browser: HTML, CSS and Javascript. We subdivided the interface in three different sections, each one composed by a triplet of files, one for each of the aforementioned technologies. The HTML files simply describes the structure of the page, and they are all fairly straightforward. As an example, below is the body of the HTML file for the game interface (*rushHour.html*).

```
<body>
    <div class="page">
        <h1>Rush Hour!</h1>
        <h2>Move vehicles to let the red car out of this madness!</h2>
        <div class="scene">
            <div class="board-container">
                <div class="board"></div>
            </div>
            <div class="button-container">
                <div class="top-row">
                    <button class="arrow-keys" id="up">   </button>
                </div>
                <div class="bottom-row">
                    <button class="arrow-keys" id="left">   </button>
                    <button class="arrow-keys" id="down">   </button>
                    <button class="arrow-keys" id="right">   </button>
                </div>
            </div>
        </div>
        <div class="logs"></div>
    </div>

</body>
```

The corresponding CSS file is simply used to style all the elements of the page, to make it more visually pleasing, but adds no functionality.

All of the logic of the page is implemented in the Javascript file *rushHour.js*. Here we have two classes

defined:

1. A Board class, representing the area in which the game takes place, initialized once each time the page is loaded. The most important field of this class is *status*, a 2D matrix made of arrays of arrays which keeps an updated status of all the cells in the board. If a cell is occupied by a vehicle, the corresponding cell in the array contains the id of the vehicle. It also has some helper methods used to set it up when the game begins, to check whether the current configuration corresponds to a win (and thus end the game), or export the state of the board to send it to the planner, which will return the next move made by Pepper.
2. A Piece class, representing one vehicle of the game. Each piece has an *id* to uniquely identify it, (the car the player has to move always has id=1), and some defining properties such as *color*, *orientation* (vertical or horizontal), *length* (which can be equal to 2 or 3, depending if the piece represents a car or a truck), and the coordinates in the board of the rightmost cell, in case of horizontal pieces, or the highest cell, in case of vertical pieces. This class too has some methods to manipulate it. The most important one is the *move* method, which is called each time the human or Pepper wants to move a piece of the board. We also used methods such as *reset* and *setBoard* to change the appearance of the board when a piece is moved, and *highlight* and *dehighlight* to change the visuals of the piece to let the player know when they selected it.

This page also features some buttons similar to the arrow keys present on every keyboard, to let the user easily move each piece on the board.

When this page first loads, a WebSocket to the main server is opened. A request to the server is made, asking for the initial configuration of the board. The server responds choosing a level based on the user's answers in the initial survey. After this, the socket is kept open, and a message is sent to the server each time the user makes a move. This message contains the updated configuration of the board. The server forwards this request to the planner, which computes a plan to let the car out of the board, and sends the first move of this plan to the server, which in turn forwards it to the interface, which finally displays Pepper's move.

Other than this, if too much time passes since the last message arrived to the interface (i.e. since Pepper made its last move), a message is sent to the server, which makes Pepper ask the user if they are having difficulties find the next move.

Below is the implementation of this socket

```
logs = document.querySelector('.logs')
let htmlBoard = document.querySelector('.board')
let board = new Board(htmlBoard)
let timer = null

// Websocket to connect to the server
ws = new WebSocket('ws://localhost:9050/websocketserver')
ws.onopen = function() {
    //ws.send('Opened web client socket')
    ws.send(JSON.stringify({id: 'gameSetup'}))
}
ws.onmessage = function(e) {
    let config = JSON.parse(e.data)
    if (config.id.includes('level')) {
        board.setup(config)
    }
    else if (config.id == 'nextMove') {
        let movingPiece = board.pieces[config.vehicle]
        for (let i=0; i<config.nCells; i++) {
            if (movingPiece.orientation == 'horizontal' && config.move_type == 'forward'
) {
```

```
                    movingPiece.move('right')
                    console.log(`Planner said: move piece ${config.vehicle} right`)
                    if (i == 0) {
                        ws.send(JSON.stringify({'motionDirection': 'right'}))
                    }
                }
                else if (movingPiece.orientation == 'horizontal' && config.move_type == '
    backward') {
                    movingPiece.move('left')
                    console.log(`Planner said: move piece ${config.vehicle} left`)
                    if (i == 0) {
                        ws.send(JSON.stringify({'motionDirection': 'left'}))
                    }
                }
                else if (movingPiece.orientation == 'vertical' && config.move_type == '
    forward') {
                    movingPiece.move('up')
                    console.log(`Planner said: move piece ${config.vehicle} up`)
                    if (i == 0) {
                        ws.send(JSON.stringify({'motionDirection': 'up'}))
                    }
                }
                else if (movingPiece.orientation == 'vertical' && config.move_type == '
    backward') {
                    movingPiece.move('down')
                    console.log(`Planner said: move piece ${config.vehicle} down`)
                    if (i == 0) {
                        ws.send(JSON.stringify({'motionDirection': 'down'}))
                    }
                }
            }
            // after one minute from the last Pepper move, Pepper asks human if they are
    struggling
            timer = window.setTimeout(() => {
                ws.send(JSON.stringify({'pepperSad': true, 'sentence': 'Are you struggling
    with your next move?'}))
            }, 1000*60) // 180 seconds
        }
    }
```

The structure of the initial survey is defined in the file called *index.HTML*. Here too, the main structure is very simple. The main logic is implemented in *index.js*. This file manages all the logic that regulate which scene to display when each button of the survey is pressed, following the flow diagram represented in Figure 4. To do so, we implemented a function called *changeScene*, which takes as input an object describing the content of the scene to display, and manages the transition between scenes. First, the text of the scene is displayed with an animation that emulates a person writing in the terminal, that happens at the same time as Peppers says it out loud. It should act as if it were subtitles to what Pepper is saying. After the text is fully displayed and Pepper as said all he had to say, the buttons corresponding to the possible answers of the user appear. When a user presses them, the *changeScene* function is called once again, now passing another object relative to the corresponding scene to be displayed, and the cycle continues, until the press of a button triggers the start of the game, in which case the interface starts displaying the game screen defined in *rushHour.HTML*.

Below is the definition of one of the objects that define a scene. Notice that the object contains information about the text to display, the text in the buttons and their color, and the functions to be called when these buttons are pressed.

```
    let welcome = {
        sceneName: 'welcome',
        text: () => 'Hello Human! Do you want to play a Rush Hour match?',
        buttons: ['Yes', 'No'],
```

```
        colors: [green, red],
        listeners: [() => {
                        ws.send(JSON.stringify({'buttonPressed': 'Yes'}))
                        changeScene(presentation)
                    },
                    () => {
                        ws.send(JSON.stringify({'buttonPressed': 'No'}))
                        changeScene(noGame)
                    }
                ]
    }
```

This file too contains a WebSocket that sends information to the main server. This happens when the user selects a the level to play, to communicate it to the server so that it can respond with the correct board configuration when the game starts, and immediately before the game interface is displayed, to send the results of the survey to the server, so that they can be recovered and updated with new information in the final survey at the end of the game.

The aforementioned final survey is described in the file, the file *survey.HTML*. The page is very simple, it contains two rows of radio buttons to let the user answer the asked questions with a rating from 1 to 5. A soon as the page loads, a WebSocket sends a request to the main server for the results that were given in the first survey, which are updated with the new answers of the users, and sent to the server once again when the window is closed and the user interaction ends. An image of this screen is shown in Figure 13.e.

5.3 Reasoning Agents

5.3.1 Use of AIPlan4EU to plan a solution of the game

Pepper's reasoning capabilities are provided and implemented through the AIPlan4EU Unified Planning framework which exploits the Unified Planning library. It allows us to solve the problem by the invocation of different automated planner i.e. *pyperplan*, *tamer*, *enhsp*, *fast-downward* and also with the possibility to use heuristics: the usage of several planners is possible thanks to the planner-independent formulation of the planning problem.

In the case of the first level

```
cars = [[1,1,2,1,1],[2,1,4,2,4],[3,3,1,4,1],[4,5,6,5,5],[5,4,3,4,2],[6,5,1,6,1],[7,5,2,6,2]]
trucks = [[8,3,4,4,4,5,4]]
vehicle, move_type, nCells = problemDefinition(cars,trucks)
if vehicle==0:
    print("No plan found")
else:
    print(f"The vehicle_{vehicle} is moving {move_type} of {nCells} number of cells")
```

the problem will have the following resulting formulation:

```
problem name = Rush Hour

types = [Cell, Vehicle]

fluents = [
  bool free_cell[fc=Cell]
  bool is_car[vehicle=Vehicle]
  bool is_truck[vehicle=Vehicle]
  bool car_at[vehicle=Vehicle, cell_1=Cell, cell_2=Cell]
  bool truck_at[vehicle=Vehicle, cell_1=Cell, cell_2=Cell, cell_3=Cell]
  bool adj_3[c1=Cell, c2=Cell, c3=Cell]
```

```
]

actions = [
  action move_forward_CAR(Vehicle v_forward_CAR, Cell c1_forward_CAR, Cell c2_forward_CAR,
    Cell c3_forward_CAR) {
    preconditions = [
      is_car(v_forward_CAR)
      car_at(v_forward_CAR, c1_forward_CAR, c2_forward_CAR)
      free_cell(c3_forward_CAR)
      adj_3(c3_forward_CAR, c1_forward_CAR, c2_forward_CAR)
    ]
    effects = [
      free_cell(c2_forward_CAR) := true
      free_cell(c3_forward_CAR) := false
      free_cell(c1_forward_CAR) := false
      car_at(v_forward_CAR, c3_forward_CAR, c1_forward_CAR) := true
      car_at(v_forward_CAR, c1_forward_CAR, c2_forward_CAR) := false
    ]
  }
  action move_backward_CAR(Vehicle v_backward_CAR, Cell c1_backward_CAR, Cell
    c2_backward_CAR, Cell c3_backward_CAR) {
    preconditions = [
      is_car(v_backward_CAR)
      car_at(v_backward_CAR, c1_backward_CAR, c2_backward_CAR)
      free_cell(c3_backward_CAR)
      adj_3(c1_backward_CAR, c2_backward_CAR, c3_backward_CAR)
    ]
    effects = [
      free_cell(c1_backward_CAR) := true
      free_cell(c2_backward_CAR) := false
      free_cell(c3_backward_CAR) := false
      car_at(v_backward_CAR, c2_backward_CAR, c3_backward_CAR) := true
      car_at(v_backward_CAR, c1_backward_CAR, c2_backward_CAR) := false
    ]
  }
  action move_forward_TRUCK(Vehicle v_forward_TRUCK, Cell c1_forward_TRUCK, Cell
    c2_forward_TRUCK, Cell c3_forward_TRUCK, Cell c4_forward_TRUCK) {
    preconditions = [
      is_truck(v_forward_TRUCK)
      truck_at(v_forward_TRUCK, c1_forward_TRUCK, c2_forward_TRUCK, c3_forward_TRUCK)
      free_cell(c4_forward_TRUCK)
      adj_3(c4_forward_TRUCK, c1_forward_TRUCK, c2_forward_TRUCK)
      adj_3(c1_forward_TRUCK, c2_forward_TRUCK, c3_forward_TRUCK)
    ]
    effects = [
      free_cell(c3_forward_TRUCK) := true
      free_cell(c4_forward_TRUCK) := false
      free_cell(c1_forward_TRUCK) := false
      free_cell(c2_forward_TRUCK) := false
      truck_at(v_forward_TRUCK, c4_forward_TRUCK, c1_forward_TRUCK, c2_forward_TRUCK) :=
    true
      truck_at(v_forward_TRUCK, c1_forward_TRUCK, c2_forward_TRUCK, c3_forward_TRUCK) :=
    false
    ]
  }
  action move_backward_TRUCK(Vehicle v_backward_TRUCK, Cell c1_backward_TRUCK, Cell
    c2_backward_TRUCK, Cell c3_backward_TRUCK, Cell c4_backward_TRUCK) {
    preconditions = [
      is_truck(v_backward_TRUCK)
      truck_at(v_backward_TRUCK, c1_backward_TRUCK, c2_backward_TRUCK, c3_backward_TRUCK)
      free_cell(c4_backward_TRUCK)
      adj_3(c1_backward_TRUCK, c2_backward_TRUCK, c3_backward_TRUCK)
      adj_3(c2_backward_TRUCK, c3_backward_TRUCK, c4_backward_TRUCK)
    ]
    effects = [
```

```
      free_cell(c1_backward_TRUCK) := true
      free_cell(c2_backward_TRUCK) := false
      free_cell(c3_backward_TRUCK) := false
      free_cell(c4_backward_TRUCK) := false
      truck_at(v_backward_TRUCK, c2_backward_TRUCK, c3_backward_TRUCK, c4_backward_TRUCK) := true
      truck_at(v_backward_TRUCK, c1_backward_TRUCK, c2_backward_TRUCK, c3_backward_TRUCK) := false
    ]
  }
]

objects = [
  Cell: [cell_11, cell_12, cell_13, cell_14, cell_15, cell_16, cell_21, ...., cell_66]
  Vehicle: [vehicle_1, vehicle_2, vehicle_3, vehicle_4, vehicle_5, vehicle_6, vehicle_7, vehicle_8]
]

initial fluents default = [
  bool free_cell[fc=Cell] := false
  bool is_car[vehicle=Vehicle] := false
  bool is_truck[vehicle=Vehicle] := false
  bool car_at[vehicle=Vehicle, cell_1=Cell, cell_2=Cell] := false
  bool truck_at[vehicle=Vehicle, cell_1=Cell, cell_2=Cell, cell_3=Cell] := false
  bool adj_3[c1=Cell, c2=Cell, c3=Cell] := false
]

initial values = [
  adj_3(cell_11, cell_21, cell_31) := true
  adj_3(cell_31, cell_21, cell_11) := true
  adj_3(cell_12, cell_22, cell_32) := true
  adj_3(cell_32, cell_22, cell_12) := true
  ...
  adj_3(cell_65, cell_55, cell_45) := true
  adj_3(cell_46, cell_56, cell_66) := true
  adj_3(cell_66, cell_56, cell_46) := true
  adj_3(cell_11, cell_12, cell_13) := true
  adj_3(cell_13, cell_12, cell_11) := true
  ...
  adj_3(cell_63, cell_64, cell_65) := true
  adj_3(cell_65, cell_64, cell_63) := true
  adj_3(cell_64, cell_65, cell_66) := true
  adj_3(cell_66, cell_65, cell_64) := true
  car_at(vehicle_1, cell_12, cell_11) := true
  is_car(vehicle_1) := true
  car_at(vehicle_2, cell_14, cell_24) := true
  is_car(vehicle_2) := true
  ...
  truck_at(vehicle_8, cell_34, cell_44, cell_54) := true
  is_truck(vehicle_8) := true
  free_cell(cell_13) := true
  free_cell(cell_15) := true
  free_cell(cell_16) := true
  ...
  free_cell(cell_64) := true
  free_cell(cell_65) := true
  free_cell(cell_66) := true
]

goals = [
  car_at(vehicle_1, cell_16, cell_15)
]
```

We have declared two types of objects: *cell* and *vehicle*. Cells are always 36 (the board is 6x6) and the

number of vehicles changes depending on the level: 9 vehicles in the first level (8 cars + 1 truck), 8 vehicles in the second level (6 cars + 2 trucks) and 5 in the third level (3 cars + 2 trucks).

Fluents are variables of the planning problem and they are quantities that can change over time. In our problem definition, we have:

- *free_ cell(x)* which checks if the cell x is free or occupied;
- *is_ car(x)* which checks if the vehicle x is a car;
- *is_ truck(x)* which checks if the vehicle x is a truck;
- *car_ at(x,y,z)* which checks if the car x is occupying the cells y and z;
- *truck_ at(x,y,z,w)* which checks if the truck x is occupying the cells y, z and w;
- *adj_ 3(x,y,z)* which checks if the three cells x, y and z are adjacent horizontally or vertically.

Note that the adjacency of the four cells in the case of the truck is checked with an AND of two *adj_ 3(x,y,z)*.

In order to describe the possible evolutions of the system, we declare 4 actions:

- *move_forward_ CAR(x,c1,c2,c3)* which allows a car to move forward. In particular cell c3 must be adjacent to cell c1 adjacent to cell c2 (see figure 6 7);
- *move_ backward_ CAR(x,c1,c2,c3)* which allows a car to move backward. in particular cell c1 must be adjacent to cell c2 adjacent to cell c3 (see figures 6 7);
- *move_forward_ TRUCK(x,c1,c2,c3,c4)* which allows a truck to move forward (the same logic of the car forward motion knowing that a truck occupies three cells);
- *move_ backward_ TRUCK(x,c1,c2,c3,c4)* which allows a truck to move backward (the same logic of the car backward motion knowing that a truck occupies three cells).

Each of these four actions is specified through preconditions and effects, defined with fluents.

Going on with the explanation, we add all the object of type *Cells* and *Vehicle* to the problem and we initialize the fluents making the "small-world assumption" i.e. it suffices to indicate the fluents that are initially true. The *free_ cell* fluent is initialized starting from the occupied cells that we know from the two set *cars* and *trucks*.

We gave a particular attention to the red car for which we save the row and the UP object since we use them to set the goal of the planning problem.

As planner, after having tested all the planners offered by the framework, we have decided to use *fast-downward* which will return the optimal path to the goal starting from the given configuration. The problem defined above is passed to this solver, that is called by the program as follows:

```
with OneshotPlanner(name='fast-downward') as planner: #fast-downward pyperplan tamer enhsp
    assert planner.supports(problem_kind)
    result = planner.solve(problem)
    if result.status == up.engines.PlanGenerationResultStatus.SOLVED_SATISFICING:
        print("Pyperplan returned: %s" % result.plan)
        completePlan = (str(result.plan)[1:-1])
        movesPlan = completePlan.split(', move_')
        move = movesPlan[0].split("(")[0][5:]
        vehicle = movesPlan[0].split("vehicle_")[1].split(",")[0]
        n_cells = 1
        for m in movesPlan[1:]:
            v = m.split("vehicle_")[1].split(",")[0]
            if move in m and vehicle==v:
```

```
                n_cells+=1
            if move not in m or vehicle!=v:
                break
        if "forward" in move:
            return vehicle, "forward", n_cells
        else:
            return vehicle, "backward", n_cells
    else:
        noPlan = "No plan found"
        return 0,noPlan,0
```

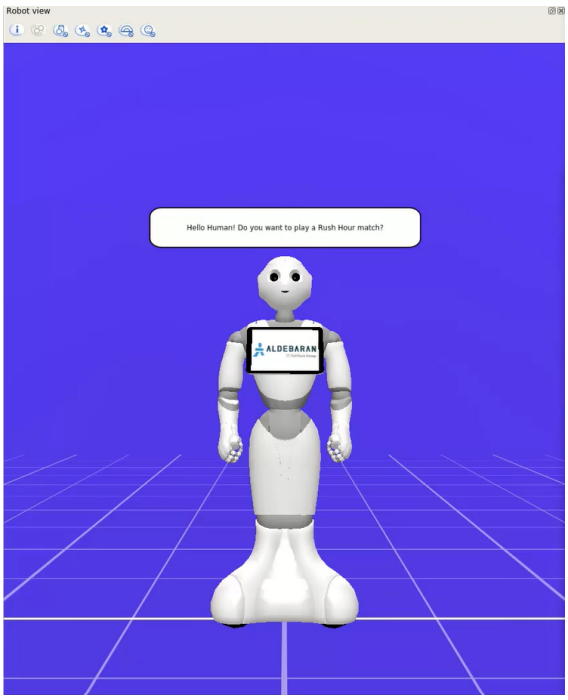
Once the planner returns the path i.e. a sequence of actions, we work on it taking only the first actions, all regarding the same vehicle and the same type of motion. To the web browser we send the id of the vehicle, the type of the motion and the number of moves in that direction and in this way the game evolves: the planner will wait for the human to take the next action and then will recompute a new plan given the new configuration.

6 Results

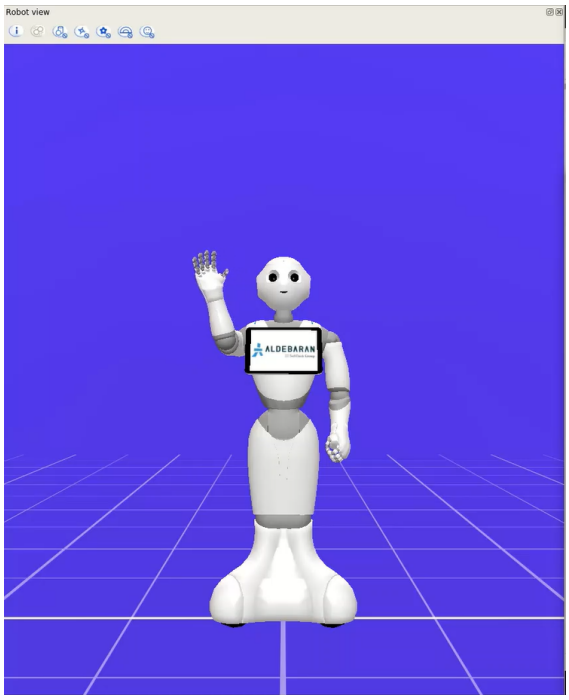
In this section all the distinct parts we presented above merge in a social and reasoning agent and the final result of our project is shown. Initially, the robot waits for a human to come near him using the sonar sensor: when the human gets close to the robot the interaction starts (figure 9). The usage of this sensor allows the robot to not invade the human personal space in the case in which the human doesn't want to begin an interaction with the robot.

At the beginning of the interaction, the connection between the simulated robot and the webpage is established; in this way, we have that the communication between Pepper and the human is instantiated on two channels: through the webpage (with the buttons and the text window) and through the human voice (this last thing is simulated writing on the Ubuntu terminal).

After the initial greetings, Pepper asks to the user if he wants to play Rush Hour game with him (fig 10): if the human refuses, Pepper becomes sad and after 15 seconds returns to check for other humans around him, otherwise Pepper asks to the human his username.

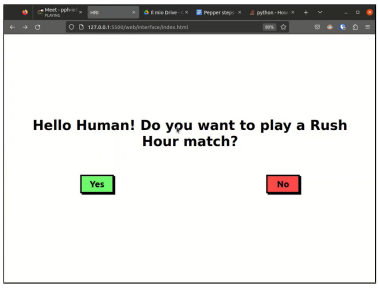


(a) What Pepper says when a human is detected.

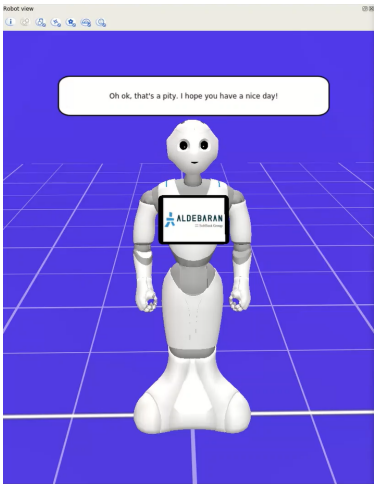


(b) Hello animation of Pepper.

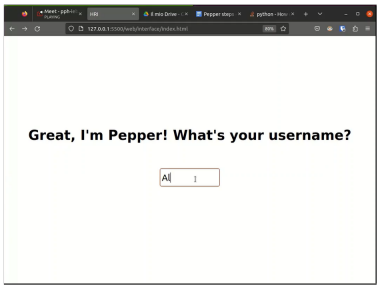
Figure 9: Initial greeting of Pepper.



(a) Match request.



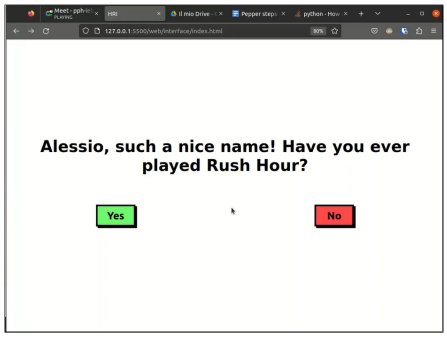
(b) Human's rejection.



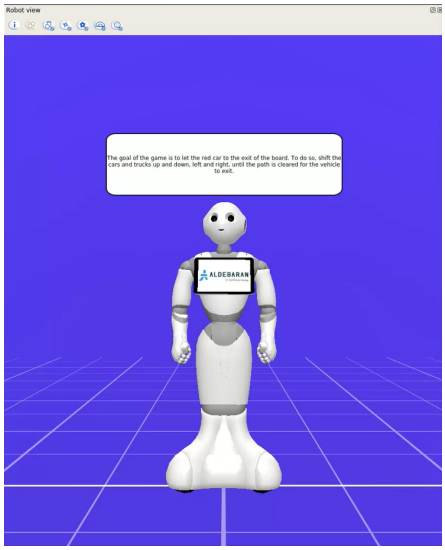
(c) Asking for the name.

Figure 10: Reactions to the first questions.

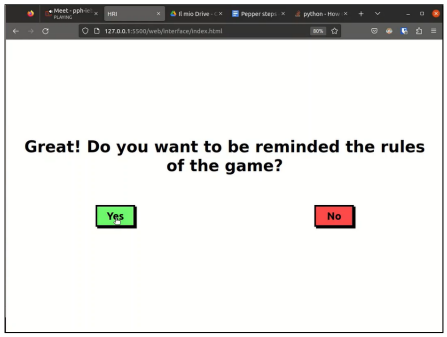
The second question that Pepper asks to the human if he has already played to Rush Hour game and if the answer is "no" Pepper explains the rules and directly selects the easy level instead if the answer is "yes" Pepper asks anyway if the human want to be remembered about the rules. In the latter case, the human will select the level by himself.



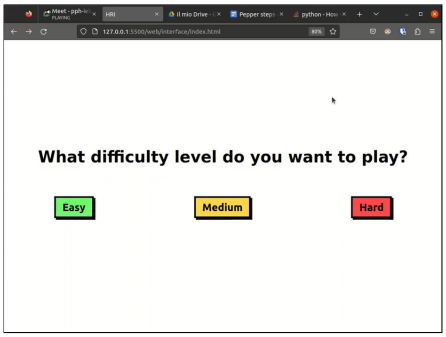
(a) Experience in Rush Hour game.



(b) Pepper's rules explanation.



(c) Rules reminding.



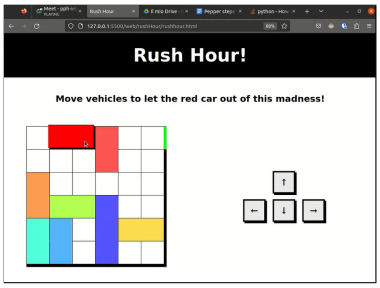
(d) Level selection.

Figure 11: Rules and level selection.

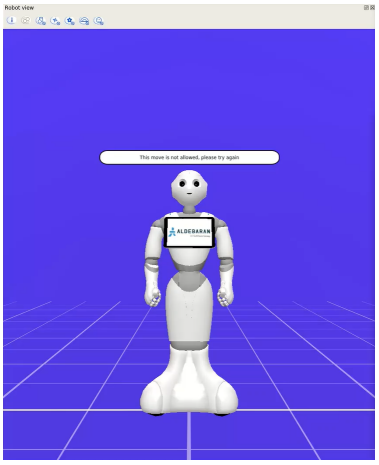
The human and the robot start playing the game in a collaborative way. Right after the human selects the level for the game, a new connection is opened between the HTML page and the Planner through the main server. The Planner, whenever receives the configuration of the game with the move made by the human, has to send a message to the main server which forwards it to the HTML in which the next move (which represents the robot next move) is specified. On the other side, there is the human which plays the game with the robot using the HTML page. At the same time, there is the robot that, based on the moves computed by the planner, receives some messages about its next moves as they are being forwarded by the main server, in order to perform animations and in order to interact with the human in the correct way. While the match is ongoing, in fact, Pepper speaks to the human and makes animations in different situations (figure 12):

- when the human performs an invalid move Pepper warns him. An invalid move is when the car or truck is moved outside the board or when is blocked by another vehicle;
- when the user is taking too much time to think about the next move, Pepper asks if he is struggling. In particular, Pepper waits 1 minute before posing the question;
- when Pepper's turn arrives, he communicates to the human that he's thinking about the next move;

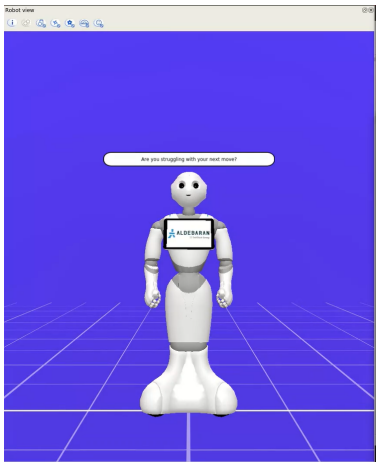
- in the first 2 situations listed above, Pepper in the meanwhile performs the animation *PepperSad*;
- when Pepper plays its move, he shifts his arm simulating the moving vehicle in the specified direction.



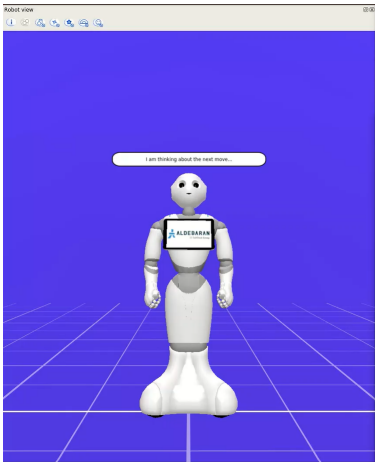
(a) Easy level configuration.



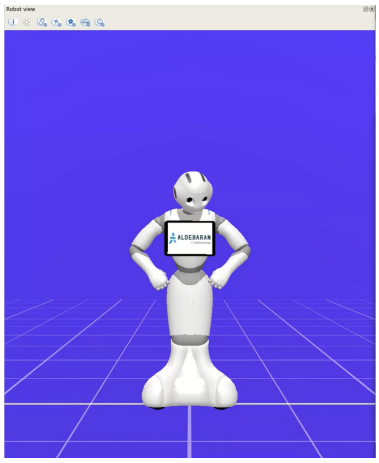
(b) Invalid move reaction.



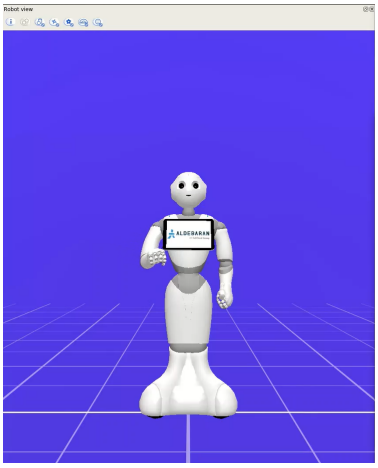
(c) Reaction to user struggle.



(d) Pepper thinking about next move.



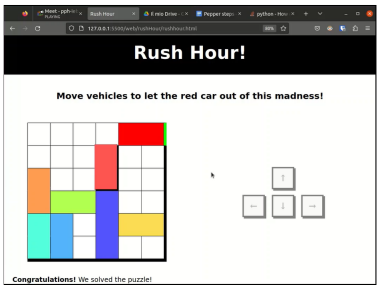
(e) Pepper sad animation.



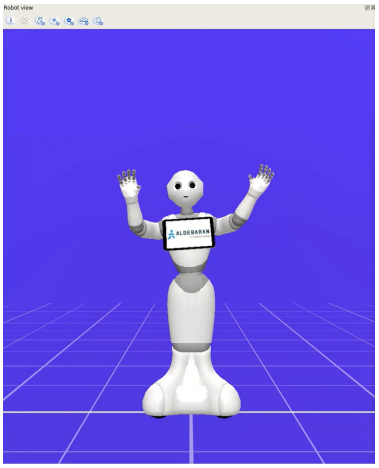
(f) Pepper move animation.

Figure 12: Additional Pepper reactions.

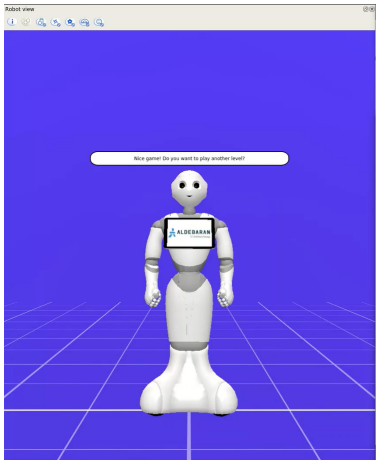
When the match ends thus the red car is out, Pepper first congratulates with the human and performs a victory dance. Then the robot asks if the human wants to play again the match at another level or if he wants to complete the final survey. In the first case, Pepper lets the human to choose the level suggesting to try with an higher level than the previous one (figure 13).



(a) Victory configuration.

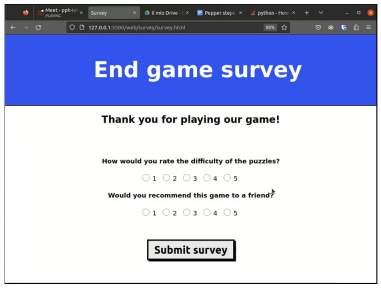


(b) Victory dance.

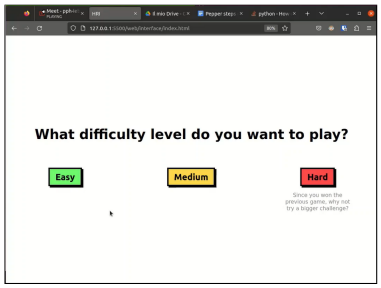


(c) New match proposal.

(d) Final choices.



(e) End game survey.



(f) Next level selection plus suggestion.

Figure 13: Final interaction steps.

The results obtained using the simulated Pepper robot are shown in a demonstrative video that is publicly available at the following link: `https://youtu.be/QCYhiQG2S14`.

7 Conclusions

In recent years we are witnessing a significant increase in the use of technologies related to robotics and artificial intelligence. The fields involved in this process and in which such technologies operate are many: ranging from entertainment for elderly people, to education for children, care and many others. It's important, in fact, to ensure that these technologies help to increase the well-being of society, in order to guarantee and pave the way for progress.
Developing this project has been very interesting from many points of view: on the one hand it was inviting to develop a communication process that was able to make the simulated robot interact with the human via an interface, on the other hand it was interesting to understand how to implement in the best possible way the robot's behavior with that of the human and therefore understand how the robot should manage its movements, gestures and dialogues in order to ensure the best possible experience for the human. The significant aspect on which we focused on is on the way in which robot is perceived and seen by human: we want the robot to be a source of help and entertainment for the user and we want also that the human feels comfortable during the interaction with the robot.

We think that our work can be further extend in different ways in both HRI and RA aspects:

- We could introduce some heuristics to make the planner faster in computing a solution to the problem. Some possible heuristics for the Rush Hour game are:
  - The distance of the red car from the exit: it is an admissible heuristic since the number of actions we need to do in order to get to a goal state is at least the distance of the red car from the exit (it could be more as we might need to move some vehicles). So this heuristic does not overestimate the actual number of actions and thus is admissible.
  - The number of cars blocking the way to the exit: it is an admissible heuristic since the number of blocking cars is lower or equal to the distance of the red car from the exit, and thus lower or equal to the number of actions we need to do.
  - The distance of the red car from the exit + the number of blocked blocking cars: it is an admissible heuristic since the actual number of steps to the goal is at least the distance of the red car from the exit (in order to drive it to the exit) + the number of blocked blocking cars as every such car adds a movement of the car that's blocking it and a movement of the car itself.
- We could test our project on a physical robot thus we could test the software part about sensing (e.g. sonars and speech) and we could use also the led of Pepper's eyes to improve the human-robot interaction.
- We could use the survey compiled by the human during the interaction with Pepper in order to make some studies. Moreover we could save the username in a database and if the human has already played Rush Hour alongside Pepper, the system would remember him having already a profile.
- We could add a facial recognition system in order to improve the interaction with the human since the robot would be able to look in the direction of the user.

In general, this project not only showcases the potential of the new technologies but also emphasizes the importance of creating robots that can engage and interact with humans in a social context. The ability of the robot to engage in a fun game like Rush Hour with a human player demonstrates the potential for robots to enhance human experiences and provide companionship, particularly for individuals who may be isolated or in need of social interaction. As we continue to advance the capabilities of robots, we can expect to see further exciting developments in the field of robotics that will continue to enhance our lives and change the way we interact with technology.

References

[1] Aiplan4eu. https://www.aiplan4eu-project.eu.

[2] Pepper. http://doc.aldebaran.com/2-5/home_pepper.html.

[3] Naoqi. http://doc.aldebaran.com/2-5/index_dev_guide.html.

[4] Choregraphe suite. http://doc.aldebaran.com/25/software/choregraphe/index.html.

[5] Tornado. https://www.tornadoweb.org/en/stable/.

[6] Christopher Fourie, Nadia Figueroa, Julie Shah, Marta Bieńkiewicz, Benoît Bardy, Etienne Burdet, Phani Teja Singamaneni, Rachid Alami, Arianna Curioni, Günther Knoblich, Wafa Johal, Dagmar Sternad, and Malte Jung. Joint action, adaptation, and entrainment in human-robot interaction. In *17th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*. 2022.

[7] Mitsuharu Matsumoto. Fragile robot: The fragility of robots induces user attachment to robots. In *European Conference On Computer Vision*. 2022.

[8] Lorenzo Cian, Talissa Dreossi, and Agostino Dovier. Modeling and solving the rush hour puzzle. In *CILC 2022: 37th Italian Conference on Computational Logic*. 2022.

[9] Ami Hauptman, Achiya Elyasaf, Moshe Sipper, and Assaf Karmon. Gp-rush: Using genetic programming to evolve solvers for the rush hour puzzle. In *11th Annual Genetic and Evolutionary Computation Conference*. 2009.