Report for Human Robot Interaction - Reasoning Agents

Vincenzo Colella 1748193 and Adriano Puglisi 1743285

October 2022

Contents

List of Figures

The authors equally contributed to the project.

1 Introduction

With the growth of relevant technologies, there is an increasing desire to deploy humanoid robots as information and service devices for people.
A few years ago, robots were only used in industrial settings where they could physically move or transport materials, the development of increasingly sophisticated robots and accurate hardware has welcomed the era of humanoid robots. People are gradually becoming acquainted with the appearance of humanoid robots that can assist, offer advice, or even take orders in a restaurant. Studies have shown that, even when robots aren't anthropomorphic, people tend to think of them as being like humans (Siino and Hinds, 2004), especially human-like robots moving around people.
Moreover, Mutlu and Forlizzi have proven that interaction failures between humans and robots are still occurring, when robots do not respond to human expectations in terms of social intelligence or act in a manner that is socially unacceptable.
This happens because people expect social robots to be more useful, meaning that they have to combine both interactions with humans and reasoning. Human-Robot interaction has been fixed thanks to many recent robots able to recognize humans and greet them, as well as talk to them and ask them questions.
A robot working in a social manner could help clients to find a specific store in a mall, the directions to a gate in an airport, walk around to advertise a new product or serve clients in a restaurant. All these examples show how robots can be used in a social context in an effective way. But these social robots are often limited to their initial goal, meaning that they are limited to few games and commands. One of the most versatile and affordable robots that accomplish the goal of being complete and social robots is Pepper, a semi-humanoid robot manufactured by SoftBank Robotics and designed with the ability to read emotions.
Pepper's capacity to perceive emotion is based on the detection and interpretation of facial expressions and speech tones.
In our work, we designed and implemented a Pepper robot that should be located at the entrance of 'Sapienza University of Rome' and that will effectively help people get to their desired building. It is incorporated with a tablet in which the user can press buttons to select whether he wants to play a memory card game or exploit the robot to have directions to a specific place inside the University. In particular, the user can simply select a building to go to or select different obstacles - places that he wants to avoid - and let the program plan an optimal path to take.

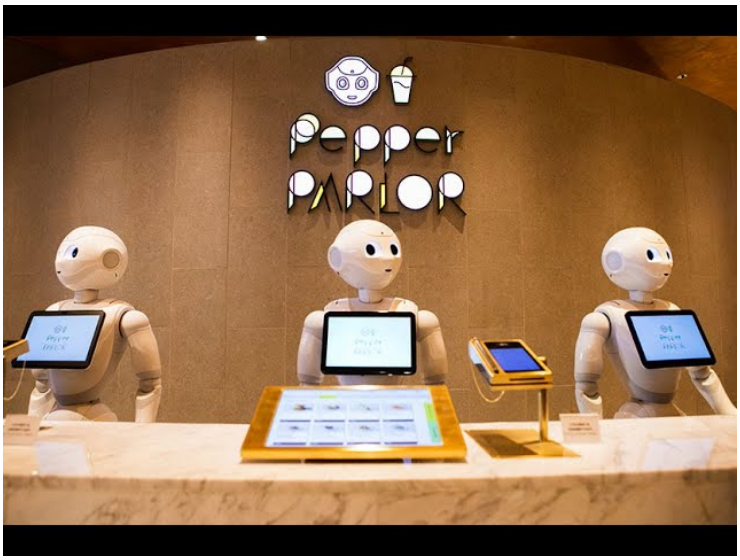


Figure 1: A collection of Pepper robots taking orders in a café named Parlor, in Shibuya, Japan.

2 Related Works

Different approaches have been tested for path planning, and different algorithms have been developed that have their own strengths and weakness.
Fast Downward is a popular heuristic search-based planning technique published in 2004. With advanced features like ADL conditions and effects and derived predicates, it can handle generic deterministic planning issues encoded in the propositional fragment of PDDL2.2 (axioms).
Fast Downward is a progression planner, seeking the space of world states of a planning assignment in the forward direction, like other well-known planners like HSP and FF. Fast Downward, however, does not directly employ the propositional PDDL model of a planning task, in contrast to other PDDL planning systems. As an alternative, multi-valued planning tasks are used to first translate the input, which makes many of the implicit limitations of a propositional planning job explicit. Fast Downward makes use of this alternate representation to compute its heuristic function, known as the causal graph heuristic, which differs significantly from typical HSP-like heuristics that ignore negative operator interactions. The three phases of Fast Downward's execution are shown in figure 2.
Another well-known algorithm widely used for path planning is A*, in which the optimal path is generated by convergence. The speed and robustness of the planned path are two key indicators of the A* algorithm's performance. There are still several flaws, such as close proximity of the path to obstructions and right-angle curves that slow down the speed. These elements cause the projected

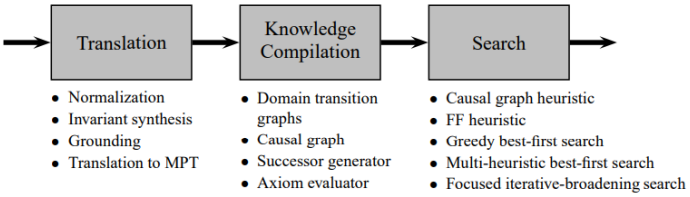


Figure 2: The three phases of Fast-Downward

path's resilience to decline. Other topics under investigation that affect the algorithm's effectiveness and the speed of the mobile robot are path planning speed and path smoothness. The suggested approach incorporates expansion distance and path smoothing to increase the resilience of the traditional A* algorithm. Since the extended nodes must no longer be traversed, the expansion distance means maintaining additional space clear of barriers to increase path dependability by preventing collisions and speeding up path planning. In some ways, reducing the map scale is equal to increasing the expansion distance. Smoothing decreases the number of right-angle turns, increasing path reliability. The suggested approach incorporates a bidirectional search strategy to speed up path design. This approach simultaneously searches from the start node and the goal node. In "The EBS-A* algorithm: An improved A* algorithm for path planning" the authors try and solve these weaknesses by implementing three strategies: expansion distance, bidirectional search optimization, and smoothing optimization. Another proposed alternative for A* is Neural A*, from Ryo

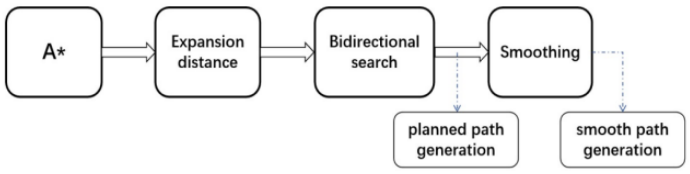


Figure 3: The process of EBS-A*

Yonetani et all. At a high level, Neural A* converts a problem instance into a guidance map using an encoder. Each node in this guidance map must pay a guidance cost. The differentiable A* module then conducts a search in accordance with the policy to reduce the overall guidance cost. The encoder learns to recognise visual signals in the training examples to increase the path accuracy and search efficiency by repeating this forward pass and the back-propagation.

Solution

3 The design

3.1 Human-Robot Interaction

We wanted to design a robot that could interact smoothly with humans, and for this reason, we started with a Pepper robot that exploits both sonar sensors and its contact sensors (on its head) to identify an approaching human.
Once the human is standing in front of Pepper, he may choose between two choices: *Game* and *Directions*. In particular, the game is a memory card game that can be played on pepper's tablet. The directions will be explained further in the next chapter.

3.2 Reasoning Agents

Concerning the reasoning agents part, our aim is to help students, staff, or any other Sapienza visitor to get directions to any building on the campus. This means that, by entering the directions menu, the user will see the 'Città Universitaria' of Sapienza in form of a grid.
From here, he can select some cells on the grid that will be marked in red, to identify 'obstacles'. These obstacles will be later avoided by the planner while finding a suitable path. Finally, when the user has selected the obstacles and decided on his final goal, the path planner will take all the data and plan the optimal plan.

3.3 Path Planner

For the path planner, we decided to use OPTIC, a temporal planner developed by King's College in London. It works by the assumption that most temporal planning techniques concentrate on reducing makespan, but makespan becomes a lower-order concern since it ignores a large class of planning issues where it is essential to take into account a greater diversity of temporal and non-temporal preferences.
In their work, J. Benton et all. analyse a basic logistics scenario where blueberries, oranges and apples must be transported to locations, B, O and A respectively. Each fruit has a distinct shelf-life. From the moment they are picked, apples last 20 days, oranges 15 days and blueberries 10 days. The truck has a long distance to travel, travelling with the perishable items from an origin point P. Let's imagine equal profit for the amount of time each product is on a shelf. The time to drive between P and B is 6 days, between P and A is 7 days, between B and O is 3 days, and between A and B is 5 days. To accomplish all deliveries, the shortest plan has a period of 15 days; that is, drive to locations A, B, and then O in that sequence.
If we were to transport the commodities in this manner, the blueberries and oranges will perish before they reach their destinations, and the total duration for

the apples would be 13 days. Instead, we need a strategy that delivers the highest total value. A strategy that drives to points B, O, and then A achieves this, however it does so in 17 days. In this scenario, adding the total time-on-shelf across all fruits gives us 15 days. To manage these circumstances, we provide two types of time-dependent goal attainment costs: those with deadlines that must be perfectly on time, as stated with PDDL3; and those with costs that grow gradually over time. Specifically, in our test situations, we employ linearly growing expenses, with a soft deadline after which cost begins to climb, and a second deadline where the whole cost is paid. If a target is reached between these time frames, the cost is calculated pro rata.
That is to say, the OPTIC developers chose to take into account modelling and reasoning with plan quality measures that are not directly associated with plan makespan. They developed a mixed integer programming encoding to handle the relationship between the hard temporal restrictions for plan stages and the soft temporal constraints for preferences. They implemented an enhancement to soft deadlines with continuous cost functions to increase the number of metrics that can be described directly in PDDL, removing the need to approximate these with various PDDL3 discrete-cost preferences.
Back to our project, when searching for the best path planner for our grid problem we found OPTIC, and by tuning in the different path adjacency costs in PDDL, we obtained appreciable results. It finds the optimal path between the entrance of the University (in which Pepper is located) and the goal, minimizing the costs, and in a good amount of time. Further details will be discussed in the implementation chapter.

4 The implementation

The project was initially created in a simulation environment using the Android Studio 4 Pepper SDK emulator.
The MODIM client connection with a local server and the Firefox browser have been used for the tablet interaction. Our GitLab repository, EAI2 project, houses all of the project's source code files as well as a video demonstration of the interaction session. *Link to the video demonstration.*
The project is composed of several modules, the most important is the main module which is responsible for the behaviour of the robot when someone is approaching or simply by running in the background waiting for a person's approach, then we have the modules for the human-robot interaction and the modules for the MODIM interaction.
Now we are going to analyze all the modules in the following sections.

4.1 Human-Robot Interaction

4.2 Main Module

The primary file that is run is *main.py*. The user will be requested to touch Pepper's head to begin the interaction once a loop is completed to wait for a user to approach, which is detected by the Class Sonar, used to monitor if somebody approached the robot.
As soon as the conversation starts, a Pepper will greet with his right hand to introduce himself.
As soon as the main function, *main_ run ()*, takes over, Pepper will ask the user what kind of interaction they would want to have by utilizing the *Speech()* class, which controls both the robot's listening and communicating.
The user can decide if he actually needs to discover a route to a given location or just wants to be entertained by the robot. Choices are made between two vocabulary terms, "Indications" or "Play," so if the user accidentally answers with one of the other terms, Pepper will declare that he did not understand and will listen to recognize the new response.

```
destinations = {"banca": "pos7_12","fisiologia umana" :"pos12_7","fisiologia generale" :"pos12_11",
                "botanica":"pos9_18","genetica":"pos10_19","medicina legale":"pos5_22","obitorio":"pos5_22",
                "scienze statistiche":"pos13_5","scienze solitiche":"pos15_9","ciao":"pos13_15",
                "lettere e filosofia":"pos16_20","scienze umanistiche":"pos16_20","chimica farmaceutica":"pos21_22",
                "laboratori chimica":"pos13_24","fisica":"pos23_14","chimica":"pos23_16","ortopedia":"pos27_15",
                "geologia":"pos18_10","giurisprudenza":"pos17_12","matematica":"pos19_22","igiene":"pos27_14",
                "zoologia":"pos21_6","neurologia":"pos24_6","scienze dello spettacolo":"pos27_8"}

vocab = ['banca', 'fisiologia', 'botanica', 'genetica', 'medicina', 'obitorio', 'scienze statistiche',
         'scienze politiche', 'ciao','lettere e filosofia', 'scienze umanistiche', 'laboratori chimica',
         'fisica', 'chimica','chimica farmaceutica', 'geologia','giurisprudenza', 'matematica',
         'igiene', 'zoologia','neurologia', 'scienze dello spettacolo', 'ortopedia']
```

Figure 4: Our dictionary and vocabulary

```
def main_run():

    interaction = start_interaction(dialogue)

    if (interaction):

        dialogue.say("Do you need indications or do you want to play?")
        rightanswer=False
        while(not rightanswer):
            answer = dialogue.listen()

            if answer == "Indications":
                rightanswer = True

                dialogue.say("Perfect! Where do you have to go?")
                goal = dialogue.listen(vocabulary=vocab)

                if (goal not in vocab):
                    find = False

                    while(not find):
                        dialogue.say("Sorry I didn't understand, can you repeat please?")
                        goal = dialogue.listen(vocabulary=vocab)

                        if (str(goal) in vocab):
                            find=True

                dialogue.say("Ok, now using the tablet press on the cells in which are obstacles, if None press the Ok button")

                #take the cell position

                for pos in destinations:
                    if (pos == goal):
                        goal = destinations[pos]

                #modify the javascript of the grid
                with open("../grid.js","r") as f:···
                with open("../grid.js","w") as f:···

                #opens interaction map
                TabletInteraction("i1")

                dialogue.say("please wait, i'm computing the best path for you")

                with open("../utils/obs.out","r") as f:···

                img_p = create_problem(obs,goal)

                with open("../actions/quit","r") as f:···

                with open("../actions/quit","w") as f:···

                dialogue.say("I'm ready! Look at my tablet to see your path")

                TabletInteraction("i3")

                dialogue.say("I hope it helped, see you next time bye!")
                gesture.sayhi()

            elif answer=="Play":
                rightanswer = True
                dialogue.say("Perfect! Let's play a memory game on the tablet")

                #opens interaction game
                TabletInteraction("i2")

                dialogue.say("I hope you had fun, see you next time bye!")
                gesture.sayhi()
            else:
                dialogue.say("Sorry, I didn't understand")
    return
```

Figure 5: Main functions

4.2.1 Directions

If the user needs directions, the interaction will continue with Pepper asking which building the user needs to go to, in this case, the vocabulary that is used includes the names of all the buildings within the campus.
Using the dictionary in which all correspondences between cell value and building are stored, the goal is replaced with its value expressed in cell number, this number is then inserted within the javascript file *grid.js* belonging to the html page *grid.html*.
Pepper will then tell the user to continue the interaction by using the tablet, in this case, it will be prompted by the html page with the grid asking the user to insert the obstacles by clicking on the cells. After this, the program will call the function *create_problem* which is responsible for the creation of the path and then always using the tablet the final route will be shown by calling the *TabletInteraction*. Once the user presses the exit button, it will be redirected to the *index.html* the interaction will stop with the robot greeting the user and the program will go back to waiting for a person to approach it.

4.2.2 Play

This part will be used only when the user doesn't need to have directions but only wants to be entertained by the robot.
In this case it will be showed the html page containing the memory game.
the user presses the exit button it will be redirected to the *index.html* page, the interaction will stop with the robot greeting the user and the program will go back to waiting for a person to approach it.

4.3 Scripts

Inside the python file *scripts_.py* are stored all the function that are used to interact with the robot:

- StartInteraction()
- class InitRobot()
- class Gestures()
- class Sonar()
- class Speech()

• **StartInteraction(speech)**:
This function is called at the beginning when a person approaches the robot, Pepper greets the user by calling the function *say()* inside the Speech() class, and then it will move his right hand simulating a greeting using the function sayhi() of the class Gestures().
• **Class InitRobot()**:
This class will initialize the robot by calling the function *setAlive()* of the file

pepper_ cmd.

• **Class Gestures()**:
Inside this class are declared four methods:

> **init() :** This method is used to initialize the class Gestures(), it calls the InitRobot class and then it defines with a list all the joint names with their corresponding integer representing the joint.
>
> **change_pose() :** This method is used to call the setPosture function stored in pepper_cmd. The current robot position, a list of joints indices whose values are desired to be altered, and a list of the new joint values are all expected as inputs for this procedure. As a result, the posture of the robot is adjusted to match these values.
>
> **head_touch() :** This method begins monitoring a head touch sensor for the predetermined period of time specified in seconds as an input parameter, and produces a Boolean result indicating whether or not the touch was detected during the monitoring period.
>
> **sayhi() :** This method handles the change of joints in order to perform the greeting, does not expect input parameters, and returns the robot's motion as output.

• **Class Sonar()**:
This class is composed of only a method called listen():

> **listen() :** The listen method in this class provides person detection. The sensor monitor uses front-sonar listening and returns the position of the person it has identified as an output method. Along with the sonar threshold, the time parameter for temporal filtering is a part of the technique parameters and may be adjusted to any desirable value.

• **Class Speech()**: This class is composed of two opposite methods, one is used to listen to the answer of the user and the other used to speak to the user

> **listen() :** The listen method is invoked, and it is repeated recursively until the robot perceives the response. This method accepts as input words and a timeout that match those of the library asr.
>
> **say() :** Contains a parameter named "answ" that is crucial for determining if the sentence the robot is speaking is intended to be a statement or an inquiry.

Another important file is *pddl2txt.py* in which are declared 3 fundamentals functions used to create the path and print it.

> **create_problem() :** This function is the one invoked from the main execution (*run_ main()*) and has as input the list containing the obstacles selected by the user and the goal which is the destination of the user. Next, it is necessary to open the file "grid_problem.pddl" in read mode

and check whether the obstacles entered are part of the free cells present (f posx_y), in case there is no particular obstacle present within the file (so the user has selected a cell that is already an obstacle), the latter is simply discarded, if it is present, the row containing the free cell corresponding to the obstacle is deleted.
The row where the goal is declared is also updated and finally, everything is written to a new pddl file "grid_problem_tmp.pddl".
Finally, the run() function is called. All this is executed if the list is not empty, otherwise the name of the image corresponding to the goal is returned as the value.
This is because if no obstacles are entered, Pepper will show the pre-calculated path to the destination.

run() : Within the run() function the pddl with the optic-clp planner is executed, when the execution is finished, the solution is cleaned of all superfluous information and is saved within a file, then the print_path() function is called

print_path():the print_path() function is used to be able to display the path, it initially reads the file where the pddl planner solution was stored, the campus image is opened and for each move, a line is drawn joining the two cells.
When the operation is completed, the new image containing the path is stored as "tmp.jpg" and the name of the image is returned in output which in this case is always "tmp.jpg"

4.4 Modim Interaction

In the file 'modim_interaction.py' we implement all functions related to the tablet interaction. This means that this code will regulate Pepper's visual support, from the Welcome Page to the Directions part.
First of all, the system will check if modim environment variable to the modim folder and throw an exception otherwise. Then the interaction class will read the input sent from the main.py function, selecting one between the three functions i1, i2 and i3. Depending on which command is sent, a different task will be executed. Each command is triggered by an interaction made by the user to Pepper by voice. When the first function is called, i1, the code will add a button 'Directions' to the Welcome Page. Then, it will display the grid page, meaning that the user asked Pepper for directions.

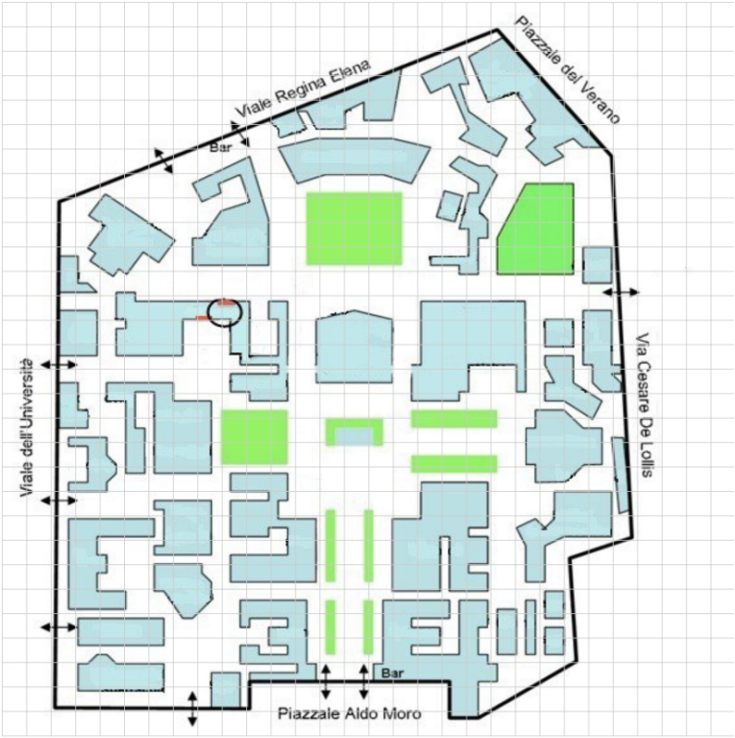


Figure 6: Sapienza University as shown on Pepper's tablet

Then the function will give time to the user to insert all obstacles that he wants to include in the map, by pressing the cells that he'd like to avoid.

At this point, we implemented a script that opens a file called obs.py and writes all the coordinates of the obstacle cells selected by the user. Specifically, to achieve this result we implemented a custom snippet in the 'grip.html' to allow us to recover the data from the obstacles. It is in fact a function called 'cell_data' that sends this data through the 'wsrobot_send' functionality once the user presses the button 'ok'.This file is fundamental and will be later used from the path planner. Later, when all the computational processes have been concluded and the optimal plan has been found, the main file will trigger the 'i3' function, which will show on the tablet the path computed and printed by internal functions of the path planner.

On the other hand, the function i2 takes care of the entertainment part, meaning that it will show the button 'game' whenever the user only wants to have fun. In particular, the code switch to the 'game.html' page in which the game is implemented.

Finally, the class Reset will restore the pages to the initial one.

```
class TabletInteraction():

    def __init__(self,do_):
        mws = ModimWSClient()
        mws.setDemoPathAuto(__file__)
        if(do_=="i1"):
            mws.run_interaction(self.i1)
        elif (do_=="i2"):mws.run_interaction(self.i2)
        else:mws.run_interaction(self.i3)

    def i1(self):
        im.init()

        a = im.ask('indications', timeout = -1)
        if(a!="timeout"):
            im.display.loadUrl('grid.html')
            a = im.ask(a,timeout=-1)
            print(os.getcwd())
            with open("/home/robot/playground/html/sample/utils/obs.out","w") as f:
                f.write(a)
                f.close()
```

Figure 7: Implementation of class Tablet Interaction along with i1 function

```
class TabletInteraction():

    def __init__(self,do_):
        mws = ModimWSClient()
        mws.setDemoPathAuto(__file__)
        if(do_=="i1"):
            mws.run_interaction(self.i1)
        elif (do_=="i2"):mws.run_interaction(self.i2)
        else:mws.run_interaction(self.i3)

    def i1(self):
        im.init()

        a = im.ask('indications', timeout = -1)
        if(a!="timeout"):
            im.display.loadUrl('grid.html')
            a = im.ask(a,timeout=-1)
            print(os.getcwd())
            with open("/home/robot/playground/html/sample/utils/obs.out","w") as f:
                f.write(a)
                f.close()
```

Figure 8: Implementation of i2 and i3 functions

4.5 Reasoning Agents

4.6 Path Planner

The implementation of the Path Planner is quite canonical, as it is a PDDL Planner in which, as said before, we exploited OPTIC. The PDDL is divided into two well-known files: the grid domain and the grid problem. It is worth explaining what the only action 'move' does: it allows the path to advance through the adjacents cells, limited by the fact that cells must be adjacent, and has the effect that one move will increase 'total-step' by one 'total-cost' by the distance. The latter will not be simply increased by one because our cost has been modelled in such a way as to push the planner to choose some cheaper cells than others. Specifically, we opted to make diagonal adjacency cost more, since without this limit, the planner would choose a zig-zag-like path as the optimal. As we just said, in the 'grid problem' file we inserted these costs for

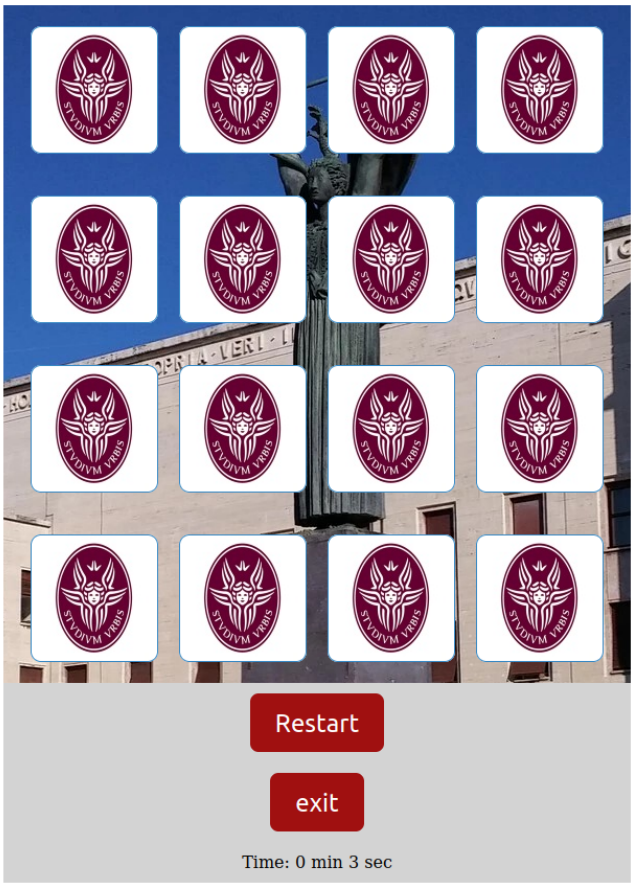


Figure 9: Memory game page

specific adjacency cells as well as the predicate f that specifies free cells in the map (those that are accessible). Finally, we wrote down all the simple adjacent cells and as the objects the positions of the cells.

5 Results

The trials were done both in simulation, using Android Studio and Pepper SDK. In order to assess the project all the different classes have been evaluated, starting with the approaching phase and then examining the interface implemented and all the conceivable behaviours. In the beginning, we have the robot waiting for someone's approach. Here it is using the class Sonar().

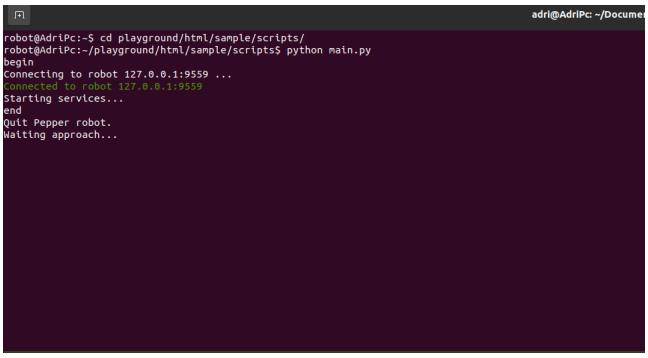


Figure 10: Initialization

5.1 Testing Directions

In this phase of testing we start interacting by simulating an approach and a touch on Pepper's head and we make the test by asking for directions. Here Pepper will also greet us by moving its right hand.

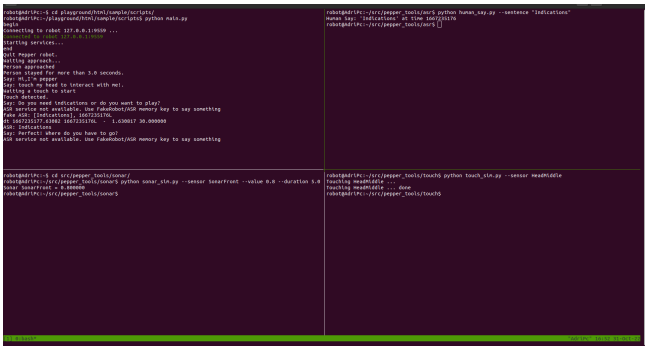
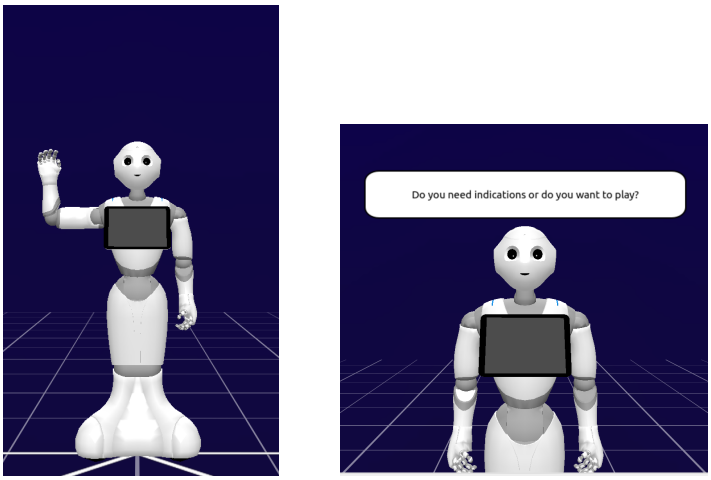


Figure 11: Directions

Then we decided to give as a goal the building "neurologia", which is on the left of the map. As expected Pepper asks the user to select the obstacles by clicking the cells on the grid of the tablet. Here, we are going to show two executions made with the same goal but with the difference that in one case we haven't



(a) Pepper Greeting (b) Pepper asking the user

Figure 12: Pepper

selected any obstacle, while in the other we decided to mark a passage in the map as an obstacle.

(a) Grid without Obstacles (b) Grid with obstacles

Figure 13: Map of Sapienza's campus with and without obstacles

At this point of testing, Pepper will compute the path by exploiting the Path Planner and will tell the user that the path is ready. It will then invite the user to look at the tablet to see the path printed. The green cell represents the 'neurologia' department, the goal.

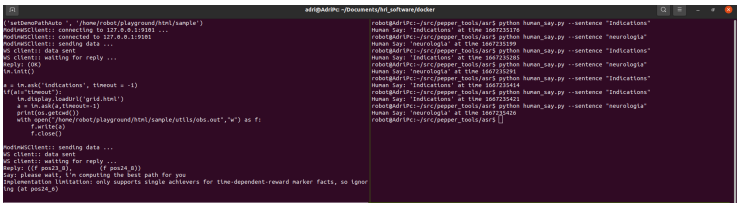


Figure 14: Pepper computing PDDL path

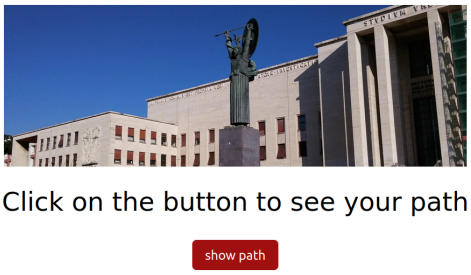


Figure 15: Path welcome page

(a) Path without obstacles (b) Path with obstacles

Figure 16: Map of Sapienza's campus with and without obstacles

5.2 Testing Game

In this trial, we tested the game implemented by us, which is a memory card game with figures containing simple symbols. First, Pepper waits for the user to say 'Play", as before. When the command is received, the button 'game' will appear on the welcome page. By pressing it, the page shown is the game, which includes a timer and two buttons to restart and exit the game. The game works by touching the tablet and trying to find all duplicates in the minimum time possible.

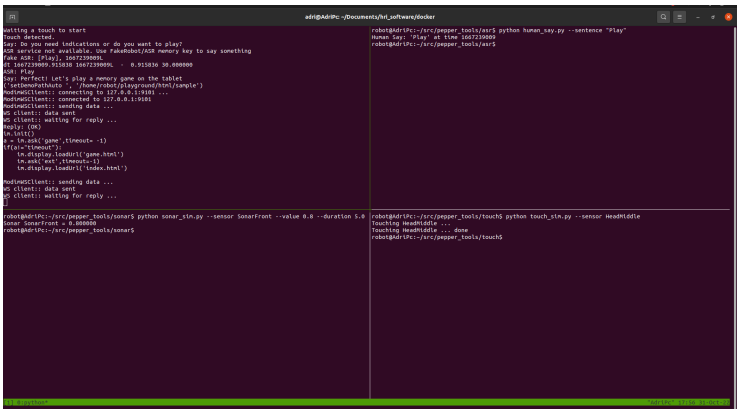


Figure 17: Game code interaction

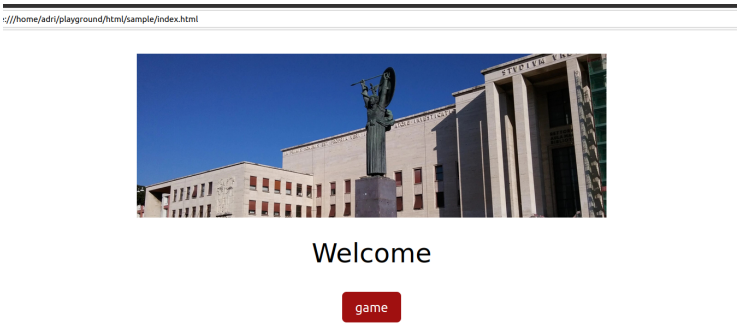


Figure 18: Game welcome page

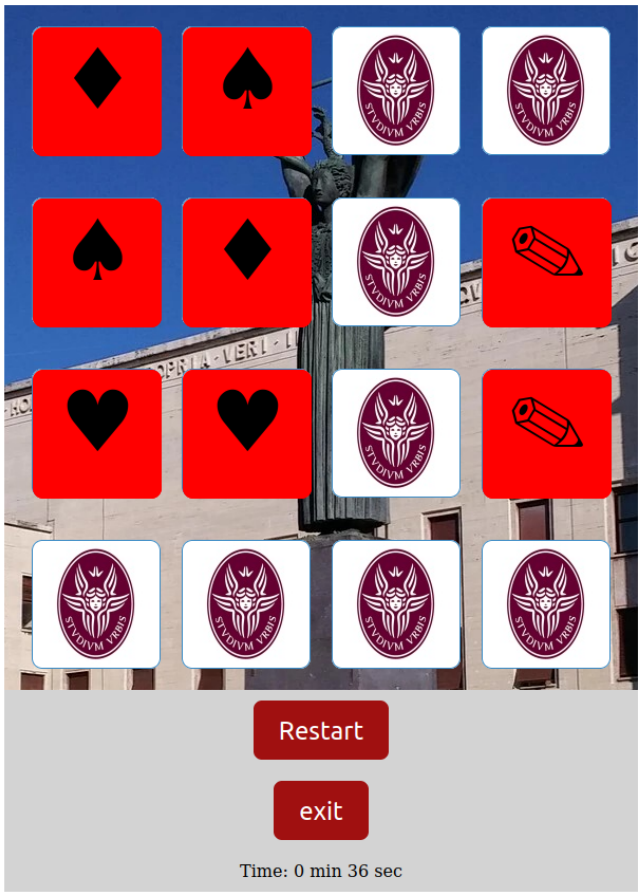


Figure 19: The memory game after few moves

6 Conclusion

During the design and implementation of this project, our aim has always been to both show how robots could interact with people in an effective way and to demonstrate that this interaction could result in useful information given to the user. That is to say that we wanted to combine, as the project required, an interaction between humans and robots that wouldn't end with futile conversations but bring this conversation to another level, in which the user can finally see through his eyes a social robot giving him important advice. Non-experts are not used to seeing a fusion between a funny robot that can hold a conversation and a what that AI that they read on the news that is able to do exceptional actions.
Here, in a scaled manner, we created a software that is able to help others in a real way and exploits what will conceivably be the future of this field: robots implemented with AIs that will surround us and help us in any possible way. Students will get accustomed to getting help from a Pepper robot, which will give them a simple path to get to a building they don't know or maybe even avoid a strike, a closed passage or a road in which workers are performing extraordinary maintenance. The appreciable results achieved show that a model like ours should be implemented at Sapienza University or any other campus and can give precious help to students and staff. Furthermore, implementing a helpful robot in an environment such as a hospital could be crucial to get relatives quickly to their loved ones.

Authors

R. M. Siino and P. J. Hinds "Robots, Gender and Sensemaking: Sex Segregation's Impact On Workers Making Sense Of a Mobile Autonomous Robot". Proceedings of the 2005 IEEE International Conference on Robotics and Automation, 2005, pp. 2773-2778, doi: 10.1109/ROBOT.2005.1570533.

Wang, H., Lou, S., Jing, J., Wang, Y., Liu, W., and Liu, T. "The EBS-A* algorithm: An improved A* algorithm for path planning". PLOS ONE, 17(2), e0263841, (2022).

Helmert, Malte "The fast downward planning system". Journal of Artificial Intelligence Research 26 (2006): 191-246.

Karur, Karthik, Nitin Sharma, Chinmay Dharmatti, and Joshua E. Siegel "A Survey of Path Planning Algorithms for Mobile Robots". Vehicles 3, no. 3: 448-468, (2021).

Yonetani, R., Taniai, T., Barekatain, M., Nishimura, M. "Path Planning using Neural A* Search". Proceedings of the 38th International Conference on Machine Learning, PMLR 139:12029-12039, 2021.

J. Benton "Temporal Planning with Preferences and Time-Dependent Continuous Costs", Dept. of Computer Science and Eng. Arizona State University Tempe, AZ 85287 USA.