Robot Learning of Social Path Planning Using a Simulator

Timo M. A. Mulder
10207511

Bachelor thesis
Credits: 18 EC

Bachelor Opleiding Kunstmatige Intelligentie

University of Amsterdam
Faculty of Science
Science Park 904
1098 XH Amsterdam

Supervisors
João T. S. Messias
Kyriacos Shiarlis
Maarten W. van Someren

Institute for Informatics
Faculty of Science
University of Amsterdam
Science Park 904
1098 XH Amsterdam

June 26th, 2015

Abstract

This project describes several considerations for designing a complete control loop with the purpose of forming a basis for simulating the learning of social navigation in robots. The robot's actions and planning are guided by a policy, which was modelled by Markov Decision Processes (MDPs), and learned from human expert example trajectories by using inverse reinforcement learning (IRL).

One of the aims of the project was to do a simple experiment where the robot tries to reach a target, and at the same time avoids a moving person in the environment. Results from the experiments look promising for future reasearch on using IRL in MDPs in the context of social navigation. Above all, the project yielded a considerable basis for conveniently conducting experiments for the larger project it was part of. This basis includes: a configured simulator, a complete description and implementation of discretising a continuous space, and a way of modelling the dynamics of a discretised environment using only small amounts of data.

Contents

Acknowledgements

Hereby I would like to thank Maarten van Someren, João Messias and Kyriacos Shiarlis for the great help, patience and support during this thesis. Special thanks goes out to Kyriacos, for the great amount of help in the last weeks of the project.

1 Introduction

Social isolation is considered a serious problem (Hortulanus et al., 2006) which is associated with poor health and loss of mobility (Wenger et al., 1996). By creating a socially intelligent semi-autonomous telepresence robot, the TERESA[1] project aims to reduce social isolation (Shiarlis et al., 2015). The main goal of the project is to give people who are physically unable to visit social events the possibility to attend these events remotely in front of their computer using TERESA.

Telepresence allows a person to feel like they are present in an environment, without the need of physically attending the location (see Figure 1a). Instead, physical parts of the human observer are presented by a machine. Telepresence varies by degree of fidelity, which in turn gives the user (i.e., the person in front of the computer) the feeling of being present to a greater or lesser extent (Sanish and Rich, 2008). For example, simple video conferencing can already give a person the feeling of being present at a location that differs from their true location. The TERESA project aims to enhance telepresence from mere video conferencing, by placing a monitor on a semi-autonomous robot (see Figure 1b). According to Shiarlis et al. (2015), early responses from participants indicate that TERESA does indeed give the feeling that the remote person is present. Moreover, participants preferred TERESA over traditional video or phone communication methods; some of them even remarked that they "saw no difference between talking with the robot or without it" (Shiarlis et al., 2015, p. 2), indicating positive prospects for the success of the TERESA project.

From preliminary findings by Shiarlis et al. (2015), it appears that, even after training, the users of TERESA encounter difficulties when controlling the robot. Also, they find it hard to determine the position and stance of the robot remotely (Shiarlis et al., 2015). As a result of this, participants experienced a reduction in the flow of the conversations (Shiarlis et al., 2015). Therefore, one of the goals of the project is to make the robot semi-autonomous.

When designing semi-autonomous actions, an important aspect to keep in mind is that TERESA's setting will mainly be social situations. This means that there will be a considerable amount of people in the environment in which the robot has to operate. As a result, traditional path planning algorithms, which aim, for example, to simply find the shortest path, do not suffice. The path planning should be enhanced with social aspects, and add these as features to the navigation. Two examples of such aspects are: keeping the robot a sufficiently large distance away from humans, and approaching humans at decent speeds. Because the amount of possible settings and therefore paths can become infeasible to describe by hand, a

[1] **TERESA**: **TE**lepresence **RE**inforcement-learning **S**ocial **A**gent. Official website: `http://teresaproject.eu/`.

(a) (b) (c)

Figure 1: (a) A user of the telepresence system at his physical location. (b) The remote location where the user is present through telerobotics. (c) The TERESA telepresence robot. Its most relevant sensors are labeled: 1 – Dalsa Genie TM HM1400; 2 – VoiceTracker TM II; 3 – Microsoft Kinect TM ; 4 – 2x Hokuyo TM UST-10LX; 5 – XSens TM MTi-30 AHRS. (Shiarlis et al., 2015, p. 1).

learning approach will be used. To be more specific, an inverse reinforcement learning (IRL) algorithm will be used to learn parts of decision functions. Section 3.2.3 will explain this algorithm.

More than just path planning, the current project aims to learn social behaviour from human examples. As pointed out earlier, normal path planning algorithms do not suffice, as the TERESA project goes beyond path planning. For each state that the robot is in, using Markov Decision Processes (MDPs), a model of the dynamics of the world, and a reward function acquired by IRL, the best action for the current state will be solved. Section 3.2 will explain this method in more detail.

Testing in a simulator is more practical than testing on the robot itself. Moreover in both reinforcement learning and IRL, often incredible amounts of trials are necessary, making it infeasible to perform these trials on a robot. Therefore, our test environment will be simulated. The basis of the simulator relies on the simulation environment Stage[2], consulted from the Robot Operating System (ROS[3]) package `stage_ros`. Sections 2.3 and 3.1 clarify why ROS and Stage are suitable for the simulation. These sections will also provide more information on the simulator and explain the configuration.

Apart from building (or configuring) a simulator, the main goal of the current project is to explore how IRL (a learning method for MDPs), can be

[2]Stage repository and documentation: `https://rtv.github.io/Stage/`

[3]ROS official website: `http://www.ros.org/`, ROS documentation (maintained by the community): `http://wiki.ros.org/`

applied to social robotic navigation. In order to do so, during this project a complete control loop will be created, which includes a simulated environment, the simulated sensors of the robot, a controller, and simulated actions for the robot (see Figure 2). Using this control loop, experiments can be conducted, which will mainly involve testing several aspects of the algorithm in the context of social navigation, including learning and planning performance. Section 1.1 gives a more detailed description of the conducted experiments. The results are presented in Section 4, and discussed in Section 5.

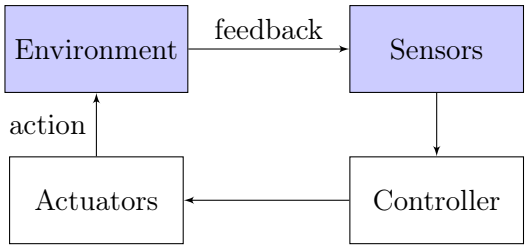


Figure 2: The complete control loop that was designed and developed during the project. The coloured blocks are dealt with by the simulator, the white blocks are solved by MDPs.

1.1 Plan for the Experimental Evaluation

The general name for the experiments that will be conducted is the *door problem*. This problem describes the situation wherein two moving subjects want to go through the same door. The subjects of the experiments are a simulated person and a simulated robot. The robot makes decisions based on a policy, which is generalised from expert policies, which will be explained in Section 3.2.

In short, the conducted experiments can be described in four steps. First, in order to learn the dynamics of the environment (as will be explained in Section 3.2.2) a random trajectory will be recorded, where as many as states are reached as possible. Subsequently, a number of expert trajectories are recorded using the simulator. The expert has to reach a target that is behind one of the outer ends of a line over which the person moves back and forth, marked with the (larger) red dot in Figure 3. The general idea is that the robot (controlled by the expert) either waits for the person to pass, or follows the person through the door. The third step is to learn parts of the model for social path planning using IRL. The final step is to evaluate the results gained from the preceding steps. The results should indicate how well the policies were learned from the expert examples. This will be evaluated by taking part of the expert trajectories for training the decision policy, and use a smaller set of trajectories for testing. To test how well the learning went, the training data will also be compared to the expert's policy.

These steps give rise to a number of sub-problems that first need to be dealt with in order to conduct the door experiment. Firstly, a complete (simulated) control loop as shown in Figure 2 needs to be created for the robot, based on the simulator and MDPs. It should take into consideration a dynamic world. Problems that need to be solved for this are: how to model the robot and the environment as an MDP in ROS? What is the best way to deal with the discrete state and action space of MDPs? How to model the dynamics of the environment? Section 3 addresses these problems. Secondly, the reward functions need to be obtained from the expert data using IRL. We need to decide what features are considered to be important, and why. This decision will be made based on experiments with the IRL algorithm.

The following section will place the current project in the context of relatively recent related work. These studies describe several important findings that are considered useful for the current project.

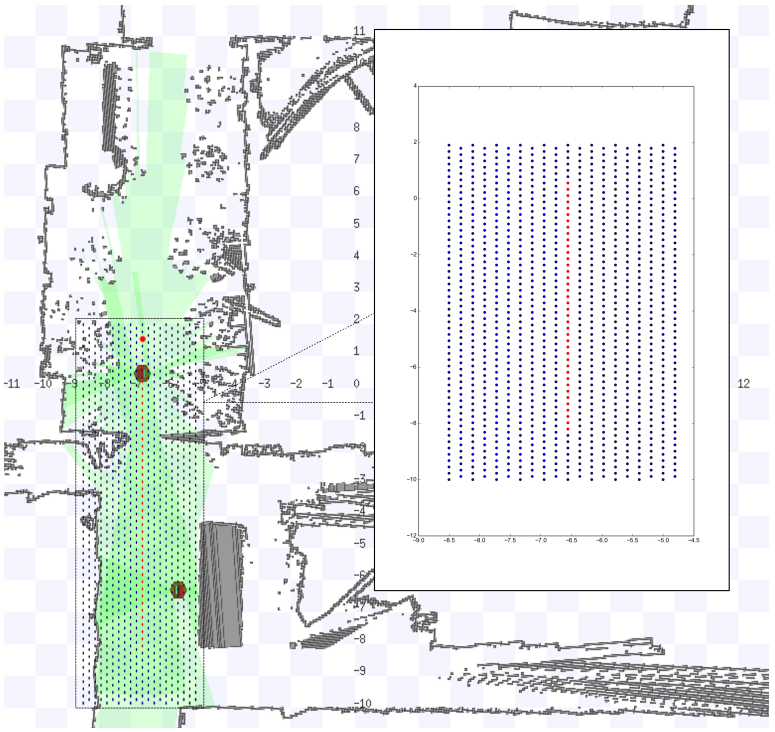


Figure 3: The door problem represented in Stage. The red dot represents the target for the robot (which is the lower brown dot). The upper brown dot, next to the red dot, represents the human at their initial position. The blue dots are the robot grid-centerpoints (as explained in Section 3.2.1), printed over the respective area that was used. The person's grid (which is a subset of the robot grid) is printed in orange. The right side of the image shows the grid centerpoints, plotted using Python.

2 Related Work

As introduced in Section 1, the experiments on social path planning are part of the TERESA project. Shiarlis et al. (2015) describe several main objectives for successfully deploying TERESA. One of these objectives is social path planning, of which the first test results are presented in Section 4. TERESA is a semi-autonomous telepresence system that is currently in development. The main goal of the project is to give physically impaired people the possibility to join social events without having to travel to the actual location (Shiarlis et al., 2015).

Having a conversation and simultaneously controlling a robot can be challenging. Moreover, it may be difficult for the controller to know the actual pose and position of the robot. Therefore, the TERESA project aims at intelligently automatising these decisions using software (Shiarlis et al., 2015). These decisions include social conversation (e.g., facing the robot screen towards the person who talks) and social navigation. As pointed out earlier, the experiments that are conducted here have the purpose of testing the present state of the social navigation algorithm that was developed for TERESA.

2.1 Path Planning

The idea of social path planning is not unique. Tipaldi and Arras (2011) already described the possible problems that robots have to cope with when they are engaged in environments with humans. Tipaldi and Arras put forward two planning problems. The first is *maximum encounter probability planning*, i.e., the shortest way to a person. The second is *minimum interference coverage*, meaning that the robot should minimise interference with people other than the target. These two planning problems describe very general goals for TERESA's social path planning.

Traditional path planning algorithms such as A* have thus far appeared to perform very well in navigation tasks (Russell and Norvig, 2010). Nevertheless, A* finds optimal paths by only taking into consideration the cost to the target and the cost to the next node. This means that in the case of a dynamic environment, to avoid collisions with obstacles and not interfere with humans, the path will need to be reconsidered often. TERESA's social path planning does not aim to calculate an optimal route by only taking the target into account as a cost function. Rather, it tries to make behavioural decisions for each state that the robot is in, taking several features (e.g., the robot's distance to a wall) of the environment into account . These decisions are modelled by Markov Decision Processes (MDPs).

Thus, instead of the strict cost function that traditional path planning algorithms use, an MDP is solved, which outputs a policy to guide the robot's behaviour. An extensive description of the particular MDPs that are solved

can be found in Section 3.2. Nevertheless, it should be mentioned that most of the implementation is based on the work by Sutton and Barto (1998), a book that provides a brief overview of topics related to reinforcement learning.

The behavioural decisions for the path planning are learned from human examples. This will be explained further in the next section.

2.2 Learning

It is considerably challenging and rather impractical to hard-code social behaviour, i.e., defining every possible action in every possible state (Shiarlis et al., 2015). Therefore, instead, the TERESA project aims to learn these behavioural decisions from examples by a human expert. One important aspect is that the behavioural policies should be applicable to numerous situations. For this reason, the policies should not be learned directly from the examples (that is, imitating the expert). Instead, the underlying reward function the expert uses to make particular decisions should be found. From the obtained reward functions, new policies are constructed using reinforcement learning.

Maximum Causal Entropy Inverse Reinforcement Learning (IRL) is an algorithm that has the capabilities to learn an underlying reward function that leads to particular decisions (Ng and Russell, 2000; Ziebart, 2010). IRL tries to find a reward function that puts forward behaviour that is close to the demonstration of the expert. Since IRL can learn the underlying patterns from human examples, it can be applied to social path planning. A more detailed explanation of this algorithm is given in Section 3.2.3. While the idea of social path planning has already been explored in the past, the use of IRL in this context is unique. Nonetheless, first results published in Shiarlis et al. (2015) and the use of IRL in contextually similar cases in the past (e.g. Henry et al., 2010) are promising.

A simulator is used to test the performance of IRL for social path planning. The next section will discuss the fundamental parts on which this simulator is based.

2.3 Simulation Environment

As pointed out in Section 1, the main task of the current project is to explore how well IRL works in the context of social path planning. Because aspects such as limited battery life make testing on the actual robot rather inconvenient, the testing of the algorithm takes place in a simulated environment.

Since useful software for our application already exists, the simulation environment is not built from scratch. Instead, the Stage simulator is used in combination with the free and open-source Robot Operating System (ROS). ROS has a modular structure, which means that different nodes can be connected to each other (Quigley et al., 2009). These nodes can publish messages to certain topics. For example, a node can send a linear movement

in x to the velocity topic of robot 1. A node can also subscribe to a certain topic, in which case it will retrieve the topic's data (Quigley et al., 2009). The modular design of ROS makes it highly configurable. Therefore, it is suitable to use in combination with the custom algorithm, discretisation and configuration of the robot.

ROS does not only provide a solid framework for the back-end, but also for the front-end. Once the configuration and the algorithm run in the back, it is relatively easy to connect these modules to the simulator. In this case, the simulator is a package called `stage_ros`, which is simply a wrapper node for the Stage simulator. According to Vaughan (2008), Stage is scalable, which means that simulating multiple robots is possible. The experiments conducted here include, among other things, testing the algorithm in combination with moving people in the environment. Thus, because of its scalability, Stage is perfectly suited for conducting experiments which involve social path planning.

3 Method

The method section is divided into two main parts, both related to individual parts of the control loop as shown in Figure 2. The first part describes the environment and the sensors of the control loop. As stated earlier, those are simulated using ROS and Stage. The simulated environment and the sensory data can be recorded, which yields training and test data that can be processed by the MDPs. The second part explains the controller and the actuators of the robot. In this part, first, a brief description on MDPs in general will be given. Secondly, a particular way of discretising the robot's and person's state and action spaces will be discussed. Thirdly, a specific way of modelling the dynamics of the world using transition functions, incorporated by the MDPs, will be explained. To solve the MDPs and acquire the generalised policies, which will in the end actuate the robot, the reward function, based on environmental features, needs to be obtained. Section 3.2.3 will discuss how IRL tries to do this. Lastly, it will be explained how the actual policy can be derived from the transition and the reward functions. The policy takes care of the path planning of the robot.

3.1 Simulator

The configuration of the simulator involved two parts. First, the simulator had to be configured, such that it was ready to use with TERESA and the experiments. Secondly, the data had to be handled correctly. The data handling refers to the input and output of the simulator, which, in terms of the control loop (as it was shown in Figure 2), connects the action to the environment, and the environment to the sensors. Therefore, first, the possibility to receive data from the simulator, for example, by recording the

data, needed to be constituted. Secondly, after the policies were created by the MDPs, the robot needed to be able to perform actions in the simulator which are concurrent with the policy.

3.1.1 Simulated Environment

Reasons such as limited battery-life and lack of processing power make it infeasible to conduct experiments regarding social path planning on the robot itself. In addition, in the real world it takes a lot of time and effort to design and create different settings in a particular environment (e.g., changing chair positions, letting people walk through the environment). Therefore, it is more convenient to simulate the environment, and conduct experiments in that simulation.

The basis of the simulator relies on Stage. Rviz can be used to visualise transformations and topics from ROS. Since there were no configuration files available to simulate TERESA properly in the simulator, these had to be created. The Stage simulator depends on configuration files with extension `.world`, which contain information about the robot's properties. The configuration file for Rviz can be created and saved by the Rviz graphical user interface (GUI). Nevertheless, making both Rviz and Stage work together properly involves more than individually configuring them.

To invoke all parts of the simulator at once, a `.launch` file was created. Launch files are ROS compatible files, written in XML, which provide functionality to consult all necessary nodes, functions, and packages at once. In this case, the launch file consults the configured Stage simulator, the configured Rviz interface, a node for automatically publishing movement actions for the person, a joystick interface to tele-operate the robot (parametrised), a controller to actuate the robot using an MDP policy (parametrised), and a publisher for the transformations of the map and simulated agents, which are received by Rviz. The launch file was configured in such a manner that more robots (e.g. to simulate humans in the environment) can be added by a simple parameter switch.

The `.world` file also contains information about the map, the robots and sensors of the robots such as laser rangefinders and kinect. The robots can be configured in this file into considerable detail. Stage automatically takes these settings into account. From the robot aspects that are configurable, four of them are now briefly discussed.

Physical aspects of the robot include: dimensions of the robot, its mass, and its locomotion. Since the TERESA robot is constituted out of different components on top of each other, it has different dimensions (in length and width) on different heights. Nevertheless, for the purpose of navigation, the most important dimensions are those at the bottom of the robot, which are 503×521 millimetre. The robot's height is 1618 mm, and it weights 10 kg. The robot has a differential locomotion, enabling it to change in orientation,

without the necessity of changing its position. Figure 4 visualises how this is possible.

Apart from the physical aspects of the robot, the way of localisation can also be configured in the `.world` configuration file. For localisation there are two options available: GPS and odometry. When the odometry option is selected, the program simply keeps track of the movements the robot makes, and determines its location using this data. While the odometry option would possibly represent the reality better, the GPS option has a significantly smaller chance of returning incorrect values. Since the experiments conducted here mainly involve a proof-of-concept for learning and deciding on the basis of human examples, it is desirable for the test-data to have minimal noise from environmental influences. Therefore, the GPS option was selected.

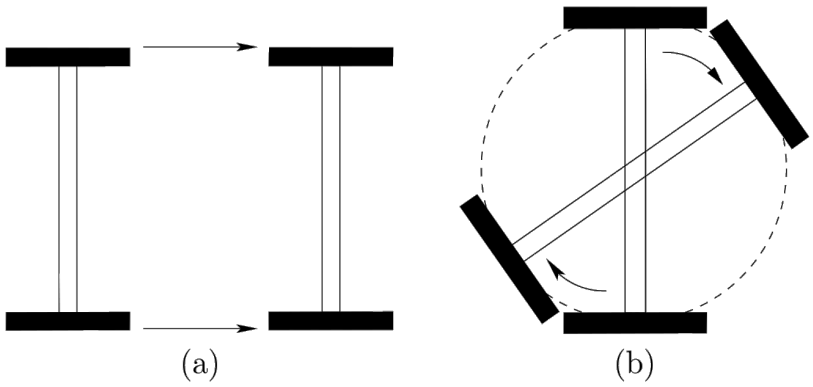


Figure 4: (a) Pure translation occurs when both wheels move at the same angular velocity; (b) pure rotation occurs when the wheels move at opposite velocities. Reprinted from *Planning Algorithms* (p. 727), by LaValle (2006).

As pointed out earlier, the `.world` file also contains information about the configuration of the map, including the conversion from pixels to meters. The pose of the map could also be defined in here, which was useful, because the map that was delivered was turned by 4.11 degrees, which is indicated by the bounding box in Figure 5a. The origin of the map is located at the centre of the horizontal x and vertical y axes.

When all preliminary parts were configured, the robot(s) could be placed in the environment. The following line of code describes how a robot is placed in the simulated world.

```
robot_name( pose [ x y z orientation ] name "reference_name")
```

Here, x, y and z represent the map coordinates. `robot_name` corresponds to the name of the previously configured robot model. The reference name can be anything, as long as it differs from other robot instances. Adding extra

robots to the Stage environment thus solely involves completing one of these lines. Nevertheless, to make the robot transformations between Stage and Rviz simultaneous, a transform publisher needs to be added to the `.launch` file, publishing the movements of the robot in Stage to Rviz.

As the experiments do not only involve avoiding static objects such as tables and walls, but rather consider a dynamic environment in which the 'obstacle' (i.e., the human) moves, a node was written that can publish simple movements to the 'person'. The reason that the word 'person' is put between quotes, is that in the simulator the person is represented as just another robot. In the end, two nodes were written. One node that simply publishes movements to make the person move in circles, and another that makes the person move back and forth over a distance of approximately 9 meter. For the experiments, only the latter was used.

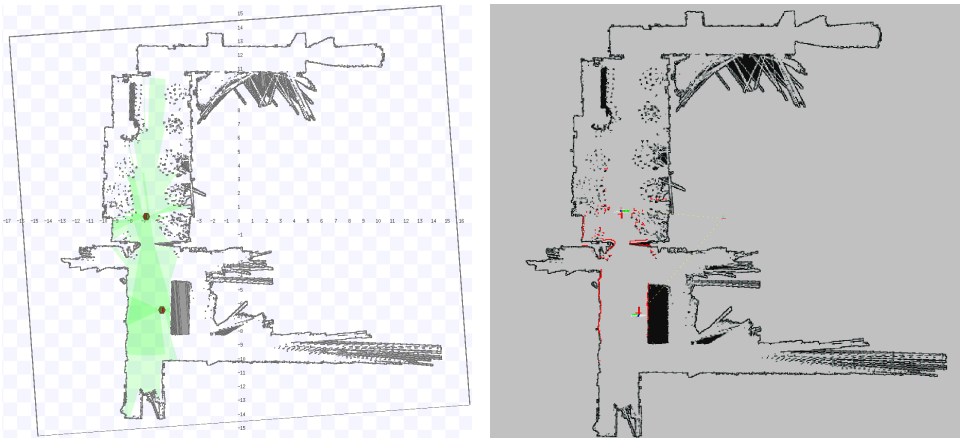
Since all previous experiments that were conducted with TERESA took place at *Les Arcades*, a nursing home in France (which is also the location where TERESA eventually is put into operation), a map of this environment, created by TERESA's laser-rangefinders, was already available. Yet, because the map was built up using laser-rangefinders, there were a lot of holes in it (this happens for example when the laser hits a window). These holes may result in bad path planning performance, since the policy may not consider a wall there, and thus a possibility to plan the path through the hole. Therefore, the map was not ready to use immediately. A fast way of closing the map is by using the free open-source image manipulation program GIMP[4]. Appendix A contains a general step-by-step description on closing maps that are created by such rangefinders.

To combine all the components of the simulator, and to make it easily installable, all the simulator elements described above were combined into a Catkin-package (Catkin is a CMake-based build system that is used to build ROS packages). Because of this, more than just being the basis of the current project's experiments, the simulator is now available to use in combination with several other upcoming experiments from the TERESA project.

The graphical user interface of the final product is shown in Figure 5. In Stage, demonstrated in Figure 5a, the green fields indicate the laser range. The brown spots are the simulated robot and person. Figure 5b shows the axes of the robot and the human. The places where the laser reflects on obstacles are also visualised, indicated by the red dots.

More than just representing the environment, the configured simulator forms a significant part in connecting the control loop together. The next section explains why the simulator yields an interface, that takes care of the control loop from the action part until (but excluding) the controller component.

[4]GNU Image Manipulation Program, `http://www.gimp.org/`



(a) The graphical user interface of the Stage simulator. A brown spot indicates a TERESA robot. For simplicity, the person has the same structure. Green coloured parts in the map indicate the range of the lasers. The grey dots next to the upper robot are chairs.

(b) The graphical user interface of Rviz. The red dots indicate that the laser has detected obstacles. The red, green and blue bars represent the axes of the robot.

Figure 5: The graphical user interfaces of the Stage and Rviz.

3.1.2 Data Handling

To collaborate with the policies that are calculated by the MDPs, more than just the general configuration of the simulator for TERESA, the possibility had to be yielded to simulate the robot's actions, obtained by solving the MDPs. Therefore, during the current project an interface was built, which is capable of handling this communication. As mentioned before, in accordance with the control loop, the interface consists out of three key parts: collecting data; discretising and processing the data; and actuating the robot using the data. The simulator is only responsible for the first and last part of the control loop, meaning that it should be able to output the environmental features and sensory data, and place the actions of the robot back into the environment.

Collecting Data: Collecting data using ROS is relatively simple. The record function from the `rosbag` package provides instant recording of all the topics one desires to record. The `.bag` file that is created by `rosbag` can be read and played using, among others, the `rqt_bag` package. Nevertheless, the data processing functions (which will be explained in Section 3.2.1) were written in Python[5]. Therefore, it was most convenient to use the `rosbag` Python API, which can read every topic parameter as detailed as required,

[5]Python 2.7.6 for Linux, `https://www.python.org/`

which in turn can be exported to any desired format that is possible within the limits of Python.

The collected data had to be processed before it was ready to use in combination with the algorithm. In this particular case, from the data collection, (x, y)-position pairs were retrieved for both the robot's and the person's movement. For the person, also an orientation was retrieved, indicating whether the person was moving up or down, or whether they were stationary. The latter was determined by the change in position, directly from the `.bag` data.

Actuation: To complete the control loop, once the MDP policies are gained from the expert data and the transition function, the policy needs to be placed back into the simulator to operate the actuators. Using the `markov_decision_making` (MDM) package by J. Messias[6], it is relatively easy to do this. The package includes an MDM library, which contains useful ROS topics such as a state and action topic for the robot. To the action topic, an action can be published, which results in the robot carrying out that particular action. To the state topic, a worldsymbol can be published. In the case of the current project, the state topic is interconnected to the discretisation functions by a module, which subscribes to the positions of the robot and the human, and returns the current global state (i.e., an index for the state of the robot, the state of the person and the orientation of the person together, see Figure 8). The MDP policy then calculates an action for the robot and publishes this to the action topic.

The global state does not only change when the robot moves, but also at every state change of the person. Therefore, it might happen that the robot is forced to reconsider its actions while it is still busy with the previous action. This had to be dealt with, as it creates the possibility for the robot to never finish an action. Therefore, a condition was built in which states that the state is only published if and only if the current robot state does not equal the previous robot state. Another problem that had to be solved is the one where the robot decides to stay. If the robot decides to stay, and it does not get updates on the states of the person, it will stay in the same state forever. Therefore, a 'fail-safe' was built, consisting of two conditions. (1) If the sum of last n states is equal to the last state times n, and the boolean `failsafe` is true: set `failsafe` to false and publish the latest state. (2) If failsafe is false, and both the last action and previous action were stagnating, publish a new state.

The next section will describe the other half of the control loop, which involves decision making and learning by processing the acquired data.

[6] `http://wiki.ros.org/markov_decision_making`

3.2 Decision Making Using MDPs

The basis for TERESA's path planning is formed by Markov Decision Processes. An MDP is defined by the four parameters: $\mathbf{S}, \mathbf{A}, \mathcal{T}$ and $\mathcal{R}$, which represent the states, the actions, a transition function and a reward function respectively. The states and actions consist out of the states and actions for both the robot and the person.

3.2.1 Discretisation

To yield the ability of making decisions, the operating space needs to be discretised into a finite set of states $\mathbf{S}$. As said, every state can consist out of multiple states. For example, the robot is in state S_r, the person is in state S_p and the orientation of the person is θ. Thus, in this case three individual *local* state pairs (S_r, S_p, θ) correspond to one *global* state $\mathbf{S}$.

A decision had to be made on the way of discretising the space. A typical grid consists of squares, where the borders of each state consist of horizontal and vertical lines. However, in the case of solving an MDP this may have significant drawbacks. The following part will explain this problem, and discuss the proposed solution.

First of all, it is expected that in a square grid, the robot can only move in four directions. This means that when the robot moves diagonally, it has to make a 'stair-like' movement, which may generally be considered to look unnatural and incompetent. For a square grid, the alternative is to extend the robot's movements to 8-DOF. However, since the probability of ending up in the desired next state may be relatively small compared to horizontal and vertical movements, the expectation is that the robot will never make use of the diagonal movements (Section 3.2.2 explains this kind of state change probabilities in more detail). Therefore, the choice was made to use a hexagonal grid, as shown in Figure 6.

In such a grid it is possible to move into six directions, with a relatively high probability of ending up in the desired state (depending on the density of the centerpoints). However, since hexagonal grids are not (very) widely used, first some discretisation functions had to be designed. Also, whereas creating a square grid and performing certain calculations on those grids is often relatively easy, it is significantly more complex to do so for a hexagonal grid. Nevertheless, in the end, all problems that had to be dealt with were solved, resulting in several Python functions, which can be found in Appendix B.

Since each state should have the same size as well as the same distance to each of its neighbouring state centerpoints, it is vital that the hexagons are regular hexagons (meaning that they are both equilateral and equiangular). Therefore, it is important that the distances along the y-axis correspond to

the distance between each centerpoint along the x-axis in the ratio,

$$x : y = \Delta : \frac{\Delta}{\sqrt{3}}. \tag{1}$$

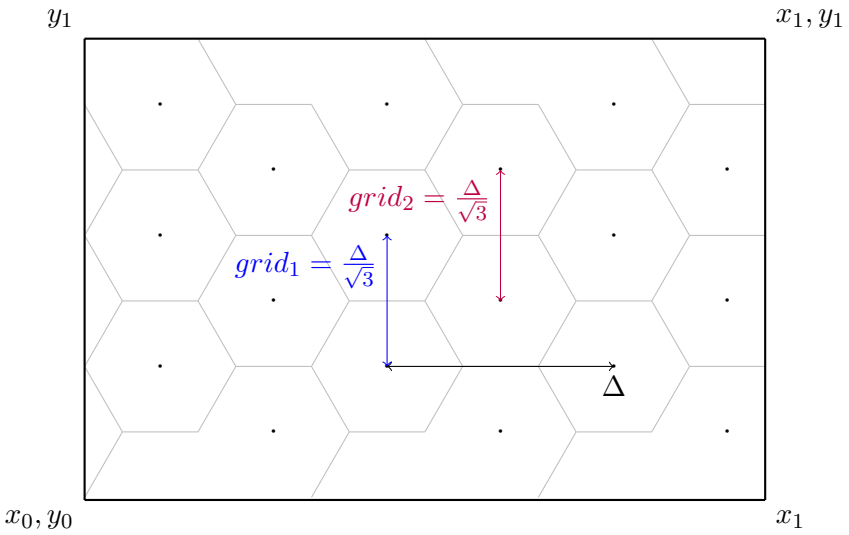


Figure 6: The hexagonal grid is built by laying two grids on top of each other, with an offset $\frac{\Delta}{2\sqrt{3}}$ in the y direction, and an offset $\frac{\Delta}{2}$ in the x direction.

In Python, using `numpy.linspace`, an x_0 and x_1 can be set, and the amount of centerpoints ϕ, that are preferred between the minimal and maximal x-value can be chosen. This results in an array containing ϕ x-values. The use of the `numpy.linspace` function makes it easy to make a grid less or more dense, which is preferable, as it creates the possibility to fit the grid properly to the input data. Too many states can lead to skipping states, whereas too few and widely distributed states do not represent the model of the world properly, and do therefore not lead to proper actions and learning. In the experiments, the values for x_0 and x_1 are -8.5 and -5.0 respectively, using 10 centerpoints per line. The y_0 and y_1 can also be chosen manually. Along the y-axis the locations of the centerpoints are automatically calculated from the ratio in Equation (1) and y_0 and y_1, where Δ is calculated from the distance between two consecutive x-centerpoints, as visualised in Figure 6. In the experiments, the range for the y-values is $[-10, 2]$.

The Cartesian product between half of the y-centerpoints and the array of x-points yields the first grid. A second line of x-values is calculated by shifting the given set of x-centerpoints by $\frac{\Delta}{2}$. From the Cartesian product with the other half of the y-values the second grid centerpoints are calculated and together, the (x, y)-pairs from both grids form the total hexagonal grid.

From this grid, a subset of states, representing the person's states, can be defined by simply giving the following values for the person: x_0, x_1, y_0, y_1.

In the experiments the respective values are: $-6.7, -6.6, -8.4, 0.7$, which were calculated from the minimum and maximum positions of the person's path. Since the person only moves up and down, this results in one column of states. The state space of both the robot and the human are shown in in Figure 3.

Because the space is discretised, the actions for the robot A_r, are discretised as well. In the experiments, the actions consist out of six possible angles, plus an 'action' do nothing. The six actions were discretised according to the formula,

$$\frac{2\pi a}{N(a) - 1} - \pi, \tag{2}$$

where a represents an action in the set $a = \{0, 1, 2, \ldots, 6\}$, and $N(a)$ is the total amount of actions. Figure 7 shows a representation of how the six actions are discretised.

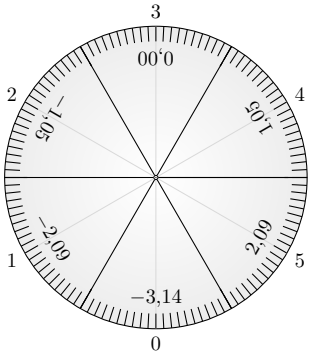


Figure 7: Classification of the robot direction into actions. The outer numbers represent the actions for the robot. The inner numbers are the angles in radians between a certain robot state S_{robot} and the next one. The black lines demarcate the borders of considering a certain angle a particular action.

The orientations of the person, which are at the same time their actions, are discretised as well. Because adding an extra dimension to the state space is costly, and is likely to significantly slow down all iteration processes, only those orientations should be included which are an absolute necessity. Because of this, and since the person only moves north and south, three different values can be assigned to the person's orientation, $\theta = \{0, 1, 2\}$. In terms of states, 0 means that the person does not move, 1 indicates an upward motion, and 2 means the person is heading towards the south. In actions, 1 and 2 are the same, only then for the person to act these motions, and 0 indicates that the person is going to stagnate.

To know in which state the person and the robot are, a function was written which calculates the nearest grid-centerpoint of an actual input

position (x, y). The nearest center is calculated by measuring the euclidean distance to all centerpoints in the appropriate grid, and then finding the argument of the minimum. These centerpoints can then be translated to the accompanying state.

As mentioned earlier, the variables $\langle S_r, S_p, \theta \rangle$ represent local states. Together, these variables form the global state. Since the amount of states grows about exponentially with the amount of steps in x, it is important to translate from local to global states in a well considered manner. In the case of the experiment, the total amount of states for the robot sum up to 1070 states. This value, times 40 states for the person, and 3 orientations, results in 128400 global states. Therefore, building a large three dimensional array, flattening it, and searching the value is an easy solution, yet quite costly. Operations such as multiplication and addition run in significant shorter time. Therefore, the following formula was used to calculate the global state:

$$\begin{aligned} \mathbf{S}_{global} = \; & S_{robot} + N\left(S_{robot}\right) \times S_{person} \\ & + N\left(S_{person}\right) \times N\left(S_{robot}\right) \times \theta \end{aligned} \tag{3}$$

where $N(S)$ is the total amount of states S. The inverse of this formula can be used to reattain the original local states. For completeness, these formulas are presented in Equations (4), (5), and (6).

$$\theta = \frac{\mathbf{S}_{global}}{N\left(S_{person}\right) \times N\left(S_{robot}\right)} \tag{4}$$

$$S_{person} = \frac{\mathbf{S}_{global} - N\left(S_{person}\right) \times N\left(S_{robot}\right) \times \theta}{N\left(S_{robot}\right)} \tag{5}$$

$$\begin{aligned} S_{robot} = \; & \mathbf{S}_{global} - \theta \times N\left(S_{person}\right) \times N\left(S_{robot}\right) \\ & - S_{human} \times N\left(S_{robot}\right) \end{aligned} \tag{6}$$

3.2.2 Modelling the Dynamics

For each time that something changes in one of the local states, the global state changes, which requires a new probability measure of going from one state to another with a certain action. These probabilities are calculated using the following transition function,

$$\mathcal{T} = \Pr\left(\mathbf{S}_{t+1} = s' \mid \mathbf{S}_t = s, \mathbf{A}_t = a\right), \tag{7}$$

which measures the probability of getting into a certain state, given the current state and actions. Given a global state, and an action, the transition function calculates what the probability is of ending up in the next state.

Therefore, the transition functions yield a model for the dynamics of the world.

In general, the transition function incorporates two types of information: states, and actions. The composition of the function may be considered rather complex. Therefore, Figure 8 gives a simplified overview of how this function is built up. For completeness, the action and state discretisation as used in the experiments are also included in this figure.

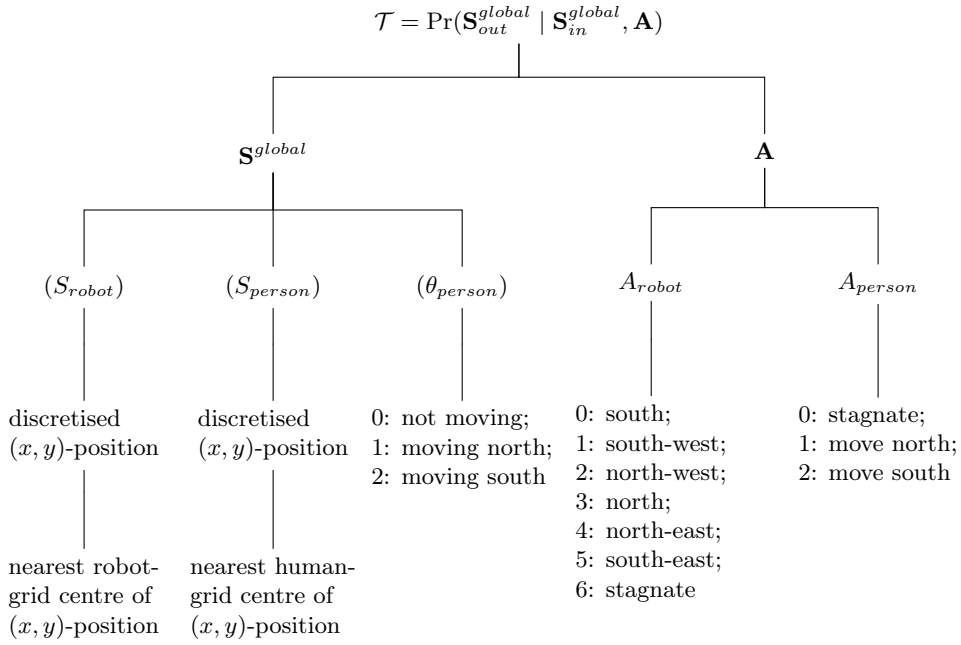


Figure 8: An overview of the transition function.

The global transition function equals the product of the outcome of the three local transition functions, including one for the robot, one for the person and one for the orientation of the person.

Doing a certain action for the robot does not always mean ending up in the state that lies in that direction. There is a slight chance to arrive in a neighbouring state, and a chance to stay in the same state (the latter especially happens when the grid is not very dense). Therefore, probabilities had to be calculated which indicate the chance of getting into a certain state with a certain action. For instance, when the robot decides to do action 2, there is a slight chance that it will actually end up in a neighbouring state in the direction 1 or 3, or it may not reach another state at all.

These possibilities were calculated, and averaged into a template. Because of the template, for each state, only those possibilities are taken into account for which the next state S_{t+1}, is a direct neighbour of the current state S. This

yields time-invariance for the transition function, as well as the opportunity to build a transition function with relatively sparse data. Section 4.1 shows the difference in probabilities by using various amounts of data.

The neighbouring states were calculated for the robot and the human separately by measuring the euclidean distance from a particular state S to all states. The states with centers that lie within a distance of $\frac{\Delta}{\sqrt{3}}$ from the center of a certain state are considered neighbours of that state. To recall, Δ defines the distance between the x-centerpoints in the grid, as shown in Figure 6. The relative location of the states can then be classified in the same manner as the robot actions, as shown in Figure 7.

To calculate the transition functions, data was recorded in which a trajectory was travelled where the aim was to cover as much states as possible. This data contains (x, y)-positions for both the human and the robot. Each time the state changed, the angle between the current and the previous state was calculated. From this, the actions could be classified, according to the borders as shown in Figure 7. For example, an action is classified as action 4 when the angle of the state change was in the range [0.5236, 1.5708].

As mentioned, from the product of the local transition probabilities, the global transitions can be calculated. A function was implemented which iterates over all the possible input states, the possible actions, and the possible output states (which are the neighbouring states of the input state and the input state itself), and returns a Python list of dictionaries. The index of each element in the list represents a global state and one of the possible actions. Thus, to retrieve the transition probabilities from a global state s, and action a in the list l, the index that needs to be requested is $a + s \times N\,(a)$, where $N(a)$ is the total amount of actions. For a certain global state and action, the corresponding dictionary can be consulted, which returns the probabilities of the transitions of getting into a particular next global state (note that this can also be the same state).

3.2.3 Learning

In an MDP, $\mathcal{R}\,(s, a)$ is the reward function, which is in this case learned by inverse reinforcement learning. In short, given a model of the environment, measurements of expert examples, and sensory input of the robot, IRL tries to extract the reward function.

IRL is a form of learning that is similar, but not equal, to *robot Learning from Demonstration* and *Apprenticeship Learning*. The difference with imitation-like learning algorithms is that instead of learning the policies directly from the observed behaviour, IRL tries to recover certain reward functions, which can then be used in a decision process (Ng and Russell, 2000). In the current project, the decision process that was used is time-invariant, making it a Markov Decision Process.

To explain further, the IRL algorithm in this case learns from human

expert example trajectories. In the case of the experiment these are 11 trajectories in which the expert tries to reach the red dot as shown in Figure 3. An expert controls the robot, and makes certain decisions. In the experiments that were conducted, these decisions include waiting for the person to come out of the door, or follow the person when the subject is headed towards the target. Inverse reinforcement learning tries to find the underlying rewards from the expert which make the expert not interfere with the person. Since the robot should learn that it should only avoid interfering with people, but not with other obstacles such as walls (the robot should not wait for the walls to move), these reward functions are rather complex. Additionally, in the experiments, the person is moving, meaning that the robot should not only reconsider its actions for every position of itself, but also for every position and orientation of the person.

Although IRL contains the words *reinforcement learning*, it should not be confused with the traditional kind of this learning type. Reinforcement learning is about *teaching* a robot certain behaviour by taking into account certain rewards for certain actions. However, IRL does not *teach* these reward functions to robots. In contrast, IRL *learns* the reward functions from expert examples. Hence the word *inverse* in IRL.

The reward function $\mathcal{R}(s, a)$ contains features $\rho(s, a)$, including: three features for the relative orientation of the person and the robot, 30 binary features which encode the distance from the robot to the nearest obstacle, 30 binary features encoding the distance from the robot to the person, and the distance from the robot to the target. The angle between the robot and the person is encoded in 16 binary features.

The reward function is defined as:

$$\mathcal{R}(s, a) = w^T \rho(s, a), \tag{8}$$

where w is a vector with weights for each feature. The features are known, however, the weights are missing. The general goal of IRL is to find the best weights for all the features, which should, with a proper transition function $\mathcal{T}$, eventually lead to optimal behavioural decisions.

3.2.4 Planning

The path planning is built by creating a policy $\pi(s, a) = \Pr(a \mid s)$ for each state. Every policy tries to maximise the overall reward of a certain path. This means that a behavioural decision is made in each state, aiming at the highest reward over the complete path. Therefore, immediate behaviour is less important than the whole for the reward. In other words, the policy plans the path from its state to the target. Because the environment is dynamic, this path is reconsidered whenever the global state changes.

After initiating the weights, using gradient descent the reward function can be fit to the expert policy values $V^{\pi}(s)$. The initial expert function is

the following:

$$V^{\pi}(s) = \sum_{h}^{H} \mathcal{R}(s_h, a_h), \tag{9}$$

since we know that $\mathcal{R}(s, a) = w^T \rho(s, a)$, this can be rewritten to:

$$V^{\pi}(s) = \sum_{h}^{H} w^T \rho(s_h, a_h) \tag{10}$$

$$= w^T \sum_{h}^{H} \rho(s_h, a_h) \tag{11}$$

Recall that the features $\rho(s, a)$ are known. Therefore, gradient descent learning can be applied to the weights w^T, which should result in a generalised reward values for the robot.

3.3 Evaluation

In general there were two types of results that had to be evaluated. One is the transition function calculation method, and one is the evaluation of the policy. The policy was evaluated in both a statistical as well as an empirical manner, as explained below.

3.3.1 Transition Generalisation

As mentioned in Section 3.2.2, the dynamics of the world were modelled by generalising the probabilities into a template, and mapping these probabilities over all the states. This yielded the possibility to gain an overall saturated transition function with relatively sparse data. The goal of this evaluation is to show that using the template-approach, not all the states need to be reached in order to measure the probabilities of the transitions well. This will be evaluated by calculating average transition tables for different amounts of data, and comparing the results.

3.3.2 Statistical Evaluation of the Policy

The generalised policy will be evaluated based on an average negative log-likelihood per decision step for each iteration of the IRL algorithm. First, the average negative log-likelihood of the expert examples given the trained policy will be calculated, without taking into account the features. The log-likelihood function is defined as:

$$\mathcal{L}(D \mid r) = \sum_{i}^{N} \sum_{t}^{T} \log\left(\pi\left(a_{i,t} \mid s_{i,t}\right)\right), \tag{12}$$

where D represents expert examples, r is the learned policy, N is the total amount of iterations, T is the total amount of time-steps, a is an action and s is a state. The average negative log-likelihood is thus:

$$\widetilde{\mathcal{L}}\left(D \mid r\right) = -\frac{\mathcal{L}\left(D \mid r\right)}{N \times T} \tag{13}$$

From this, the new policy can be evaluated by leaving three example trajectories out of the training data, and using these as test trajectories. Again, the average negative log-likelihood can be calculated in the same manner as in Equation (12) and (13), in this case comparing the test trajectories with the trained policy. Nevertheless, the focus here lies on testing how well the policy can generalise, which should indicate how good the decisions are.

Since there are seven actions for each state, the decisions are considered random if they are correct in the ratio $1 : 7$. Therefore, the average negative log-likelihood of making random decisions is $\ln\left(7\right) \approx 1.95$. The expectations are that the test where the features (e.g., the distance from the robot to the walls, the relative orientation between the robot and the person) were not taken into account will perform equally well as random, since it can only learn the states it has seen before. Thus, without features, there is very little chance that the policy can generalise. In contrast, by using correctly initialised weights for the features, learning should be possible. This should produce a lower average negative log-likelihood than taking random actions.

The results of these evaluations will be presented in a graph, showing the average likelihood per decision step for each iteration, for both the training and test-set, both with and without features.

3.3.3 Empirical Evaluation

To evaluate the policy in an empirical way, it was transformed to `Boost I/O` functions, so that it can work with the ROS `mdm_library`. This way, using the simulator, a visualised representation of the robot's actions could be attained.

To evaluate the decisions of the robot, the simulator, where the robot was controlled by the policy, was run six times, for five different starting locations. Thus, in total, there were 30 runs of the simulator. For every starting position, the averages of the outcomes of the following two simple observations were evaluated:

- Did the robot reach the target?
- Did the robot not interfere with the person (i.e., did it not collide with the person)?

For each starting position, which is at least 1 meter away from all the other starting positions, the percentage of achieving the objective was calculated.

Two types of policies were tested. One policy where the features were taken into account, and one without features. As pointed out earlier, the policy without features is prone to make random decisions unless it has seen the state before. Therefore, a comparison between the two policies should indicate path planning performance.

4 Results

4.1 Transition Generalisation

The following results show the difference between transition templates that were based on different amounts of data ($N = 100, N = 250, N = 500, N = 1000$ and $N = 1241$). First, the Tables 1–6 represent the robot transitions for each N. The tables for the person are shown in Tables 7–12. Red filled cells mean an undesired value for that cell. Grey coloured cells, in the case of the robot the diagonals, indicate places where the direction of the action and the direction of the next state were the same. The first column of each table describes the actions, and the first row the output state.

	0	**1**	**2**	**3**	**4**	**5**	**self**
0	0.46	0.10	0.00	0.00	0.00	0.13	0.31
1	0.00	0.40	0.00	0.00	0.00	0.00	0.60
2	0.00	0.00	0.50	0.00	0.00	0.00	0.50
3	0.00	0.00	0.17	0.36	0.19	0.00	0.29
4	0.00	0.00	0.00	0.00	0.00	0.00	0.00
5	0.00	0.00	0.00	0.00	0.00	0.50	0.50
stay	0.00	0.00	0.00	0.00	0.00	0.00	1.00

Table 1: $\mathcal{T}_{robot}$, $N = 100$.

	0	**1**	**2**	**3**	**4**	**5**	**self**
0	0.43	0.12	0.00	0.00	0.00	0.13	0.31
1	0.13	0.40	0.13	0.00	0.00	0.00	0.33
2	0.00	0.18	0.23	0.18	0.00	0.00	0.41
3	0.00	0.00	0.15	0.46	0.10	0.00	0.29
4	0.00	0.00	0.00	0.17	0.33	0.17	0.33
5	0.14	0.00	0.00	0.00	0.05	0.52	0.29
stay	0.00	0.00	0.00	0.00	0.00	0.00	1.00

Table 2: $\mathcal{T}_{robot}$, $N = 250$.

	0	**1**	**2**	**3**	**4**	**5**	**self**
0	0.41	0.15	0.00	0.00	0.00	0.15	0.29
1	0.13	0.39	0.10	0.00	0.00	0.00	0.39
2	0.00	0.18	0.20	0.24	0.00	0.00	0.38
3	0.00	0.00	0.16	0.44	0.11	0.00	0.28
4	0.00	0.00	0.00	0.08	0.42	0.08	0.42
5	0.15	0.00	0.00	0.00	0.12	0.41	0.32
stay	0.00	0.00	0.00	0.00	0.00	0.00	1.00

Table 3: $\mathcal{T}_{robot}$, $N = 500$.

	0	**1**	**2**	**3**	**4**	**5**	**self**
0	0.39	0.17	0.00	0.00	0.00	0.17	0.26
1	0.17	0.26	0.22	0.00	0.00	0.00	0.35
2	0.00	0.17	0.25	0.22	0.00	0.00	0.36
3	0.00	0.00	0.16	0.42	0.14	0.00	0.28
4	0.00	0.00	0.00	0.12	0.38	0.10	0.40
5	0.13	0.00	0.00	0.00	0.11	0.43	0.33
stay	0.00	0.00	0.00	0.00	0.00	0.00	1.00

Table 4: $\mathcal{T}_{robot}$, $N = 750$.

	0	**1**	**2**	**3**	**4**	**5**	**self**
0	0.39	0.16	0.00	0.00	0.00	0.17	0.28
1	0.18	0.29	0.20	0.00	0.00	0.00	0.33
2	0.00	0.15	0.27	0.23	0.00	0.00	0.35
3	0.00	0.00	0.16	0.42	0.13	0.00	0.29
4	0.00	0.00	0.00	0.13	0.38	0.10	0.38
5	0.16	0.00	0.00	0.00	0.15	0.37	0.32
stay	0.00	0.00	0.00	0.00	0.00	0.00	1.00

Table 5: $\mathcal{T}_{robot}$, $N = 1000$.

	0	**1**	**2**	**3**	**4**	**5**	**self**
0	0.38	0.16	0.00	0.00	0.00	0.17	0.29
1	0.21	0.30	0.19	0.00	0.00	0.00	0.31
2	0.00	0.13	0.30	0.23	0.00	0.00	0.35
3	0.00	0.00	0.17	0.39	0.15	0.00	0.28
4	0.00	0.00	0.00	0.15	0.40	0.10	0.35
5	0.19	0.00	0.00	0.00	0.12	0.38	0.31
stay	0.00	0.00	0.00	0.00	0.00	0.00	1.00

Table 6: $\mathcal{T}_{robot}$, $N = 1241$.

	0	1	2	3	4	5	same
0	0.00	0.00	0.00	0.00	0.00	0.00	1.00
1	0.00	0.00	0.00	0.29	0.00	0.00	0.71
2	0.44	0.00	0.00	0.01	0.00	0.00	0.55

Table 7: $\mathcal{T}_{person}$, $N = 100$.

	0	1	2	3	4	5	same
0	0.00	0.00	0.00	0.00	0.00	0.00	1.00
1	0.00	0.00	0.00	0.39	0.00	0.00	0.61
2	0.43	0.00	0.00	0.01	0.00	0.00	0.56

Table 8: $\mathcal{T}_{person}$ $N = 250$.

	0	1	2	3	4	5	same
0	0.00	0.00	0.00	0.00	0.00	0.00	1.00
1	0.00	0.00	0.00	0.38	0.00	0.00	0.62
2	0.44	0.00	0.00	0.00	0.00	0.00	0.56

Table 9: $\mathcal{T}_{person}$, $N = 500$.

	0	1	2	3	4	5	same
0	0.00	0.00	0.00	0.00	0.00	0.00	1.00
1	0.00	0.00	0.00	0.41	0.00	0.00	0.59
2	0.42	0.00	0.00	0.00	0.00	0.00	0.58

Table 10: $\mathcal{T}_{person}$, $N = 750$.

	0	1	2	3	4	5	same
0	0.00	0.00	0.00	0.00	0.00	0.00	1.00
1	0.00	0.00	0.00	0.40	0.00	0.00	0.60
2	0.41	0.00	0.00	0.00	0.00	0.00	0.59

Table 11: $\mathcal{T}_{person}$, $N = 1000$.

	0	1	2	3	4	5	same
0	0.00	0.00	0.00	0.00	0.00	0.00	1.00
1	0.00	0.00	0.00	0.40	0.00	0.00	0.60
2	0.40	0.00	0.00	0.00	0.00	0.00	0.60

Table 12: $\mathcal{T}_{person}$, $N = 1241$.

4.2 Results of the Policy

The average negative log-likelihoods, calculated as described in Section 3.3.2 are presented in Figure 9, and will be discussed below.

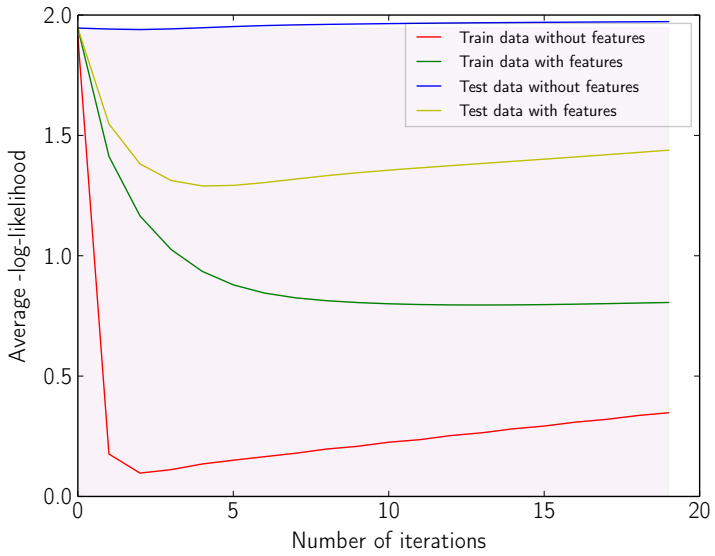


Figure 9: The average negative log-likelihood per decision step for each iteration. The number of iterations are printed along the horizontal-axis; the average log-likelihood along the vertical-axis. The field under ln(7) is shaded. Values that lie beneath this value are better than random.

The results of plugging the policies back into the simulator are shown in Figure 10. The results are based on the two objectives, as constituted in Section 3.3.3. The differences in results will be discussed in Section 5.2.

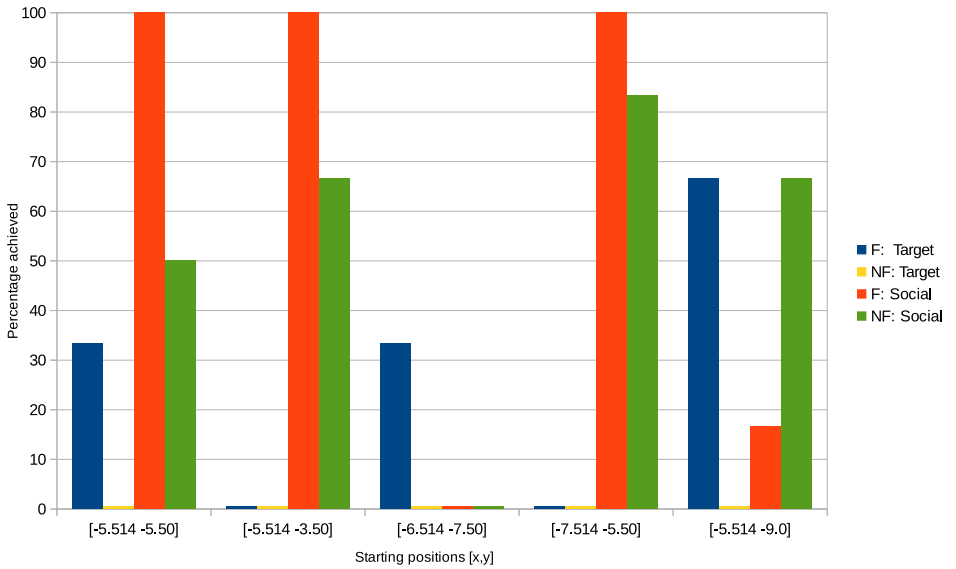


Figure 10: The results of the observed behaviour. In the legend, F indicates the policy that took features into account. NF indicates results from the policy without feature training. On the horizontal-axis the starting positions, which were used by the simulator, are presented in $[x, y]$.

5 Discussion

In the following sections, first, the results that involve modelling the environment will be evaluated. Subsequently, the policy performance will be discussed for two types of results: statistical and empirical. The whole will be concluded by a general comment, reflecting on the produced work.

5.1 Transition Generalisation

Tables 1–6 in Section 4.1 show the transition tables for the robot. From a sample size of 250 state changes onwards, relatively little difference in the key values can be found. The key values are indicated with a grey shade, which represents the corresponding direction of a certain action. These values are generally larger than the other values for that particular action. Nevertheless, in Table 1, no values could be obtained for action 4. This can be explained by the fact that the recording of the data started in a position where there was no possibility to go right.

The probability of staying in the same state is 31.5% on average for the largest dataset. This could indicate that the grid was not very dense, and therefore the robot stayed in the same state, even though it moved. However, Figure 3 shows a relatively dense grid compared to the space it covers. This can be explained by the fact that a state change was registered each time the global state changed. This means that the state also changes when the person's state changes, which might result in unfinished actions. To solve this problem, a solution could be to make the grid centerpoints more dense. On the other hand, as mentioned before, the amount of states, and therefore the calculations, grow exponentially with the amount of centerpoints in the x-direction of the grid. Furthermore, the transition functions were calculated to solve this problem in the first place, by taking these probabilities into account.

The template-approach for modelling the dynamics of the environment shows good results, as it creates the possibility of building a model like this in a relatively short period of time. For comparison, $N = 1241$ are the amount of states (filtered on state change) that were gained using only 445 seconds of data.

The results for the person, shown in Tables 7–12, show similar results to the one of the robot. From 250 samples onwards the probability distributions seem relatively stable, which means that enough states and actions were seen to generalise the transition function.

5.2 Evaluation of the Policy

The red line in Figure 9 represents the average negative log-likelihood of the training data without features compared to the expert examples. Because no features were learned, these results are highly dependent on the working of a transition function. As the graph shows, the resulting policy is very similar to the expert examples, which suggests an over-fit. At the same time, it indicates a proper working of the transition function. However, because of the over-fit, it is not possible to generalise from this data, as shown by the blue line, which represents the test data without features. The average negative log-likelihood of the test-data contains values which are close to those indicating random actions.

Most likely because a momentum gradient descent ($\alpha_t + \epsilon\alpha_{t-1}$) method was used in the algorithm, the trained policy without features almost converged at 2 iterations. However, it then goes up again, because it still has the momentum ϵ of the previous iteration.

The green line represents the trained policy with features compared with the expert examples. The average negative log-likelihood values of the training set with features is worse than the training data set without features. Nevertheless, the test data shows that by using the features the policy is indeed better generalised, indicating a better learning performance.

To test the performance of the yielded policies in a less statistical and more practical way, both policies were plugged back in the simulator, to see their performance and compare them. As explained in Section 3.3.3, two easy to register criteria were measured, which included reaching the target, and performing some social behaviour. The resulting graph is shown in Figure 10. Along the horizontal-axis, the figure shows the results, ordered by each of the starting positions in $[x, y]$. Along the vertical-axis, the percentage of success is shown per objective, per policy.

As expected, without learning the features, the robot carried out random actions, which did not look intelligent at all. Because of these random actions, it bumped into a wall in all test samples before reaching the target, as indicated by the values of 'NF: Target' in Figure 10 (which is always zero percent).

As shown in Figure 3, states were also considered which were actually walls. Therefore, even thought the features for the wall should prevent the robot from going there, the planner takes into account these states, resulting in a collision with the wall in some of the runs. This was especially the case when the robot started relatively close to a wall, as happened from starting position $[-5.514, -3.50]$, where its first action would then be to go up, resulting in an accident immediately.

Using the policy with trained features, the overall social behaviour was somewhat better than in the one without features. In the trajectory with starting position $[-5.514, -9.0]$, the robot reached the target most often. However, unfortunately, this was at the expense of social behaviour, as it interfered with the person in five out of six runs.

The graph shows that the performance in social behaviour using the trained features is better than the performance of the policy without features in all cases except for the last one. This points towards a positive prospect for the robot to learn social behaviour. As the graph shows, in all the samples from three out of five trajectories, the robot did not interfere with the person. Nevertheless, more extensive and intensive experiments should be conducted to know if the results are significant.

Determining the optimal parameters to initialise the IRL algorithm with could improve the results significantly. Nevertheless, adjusting these parameters is often time consuming, as it has been for the results that were achieved, and further adjustment was thus outside of the project's scope.

In general the current project yields a considerable basis for conducting future experiments in the TERESA project. Things that were designed and created include a fully configured simulator, a way of discretising a continuous space into a finite set of states and actions as well as an implementation of that, a template-approach of a transition function, some initial results on the working of IRL and MDPs in the context of social path planning, and a complete simulated environment in order to test resulting policies in a convenient way. Initial results, in which the performance of decision making

on the basis of policies that were modelled by MDPs and IRL were tested, seem promising. Nevertheless, considerable adjustments need to be made in order to use these policies in the real world.

References

Henry, P., Vollmer, C., Ferris, B., and Fox, D. (2010). Learning to navigate through crowded environments. In *Robotics and Automation (ICRA), IEEE International Conference on*, pages 981–986. IEEE.

Hortulanus, R., Machielse, A., and Meeuwesen, L. (2006). *Social isolation in modern society*, volume 10. Routledge.

LaValle, S. M. (2006). *Planning algorithms*. Cambridge university press.

Ng, A. Y. and Russell, S. J. (2000). Algorithms for inverse reinforcement learning. In *International Conference on Machine Learning*, pages 663–670.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5.

Russell, S. and Norvig, P. (2010). *Artificial Intelligence: A modern approach International Edition*. Upper Saddle River, NJ: Pearson, third edition.

Sanish, K. B. and Rich, C. (2008). Dataquest insight: The telepresence market in india, 2008. *Gartner*.

Shiarlis, K., Messias, J., van Someren, M., Whiteson, S., Kim, J., Vroon, J., Englebienne, G., Truong, K., Evers, V., Pérez-Higueras, N., et al. (March, 2015). Teresa: A socially intelligent semi-autonomous telepresence system. In *Robotics and Automation (ICRA), IEEE International Conference on*.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

Tipaldi, G. D. and Arras, K. O. (2011). Planning problems for social robots. In *International Conference on Automated Planning and Scheduling*.

Vaughan, R. (2008). Massively multi-robot simulation in stage. *Swarm Intelligence*, 2(2-4):189–208.

Wenger, G. C., Davies, R., Shahtahmasebi, S., and Scott, A. (1996). Social isolation and loneliness in old age: Review and model refinement. *Ageing and Society*, 16(03):333–358.

Ziebart, B. D. (2010). Modeling purposeful adaptive behavior with the principle of maximum causal entropy. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 1255–1262.

Appendices

A Step-by-step Guide on Closing a Map

Maps of robot operating environments can conveniently be created by robot laser-rangefinders. Nevertheless, these maps are not always 'closed', that is, parts of the walls are missing, possibly because of a window. Therefore, this appendix contains a step-by-step guide on how to close the map using GIMP[7] 2.8 for Linux.

1. Open the pgm file containing the map in GIMP
2. Go to *Colors* and click on *Threshold...*
3. Slide from left black until 206 (or how far necessary to change all grey into black) and click *apply*
4. From the tool menu (the one with all the icons) click on fuzzy select tool and select the black outerpart
5. Right click in the image → *Select* → *Shrink..* → 1px → click *OK*
6. Press delete key on your keyboard
7. From the menu bar, select *File* → *Export as...*
8. Select pgm RAW as file format (or any other format that you prefer)
9. *Export*

[7] `http://www.gimp.org`

B Python Key Functions to Calculate Transition Functions in a Hexagonal State-space

def nearestCenter(self, position_vector, human):

This function calculates the nearest grid centerpoint of an input `[x,y]`. The boolean parameter `human` changes the grid to either the human grid or the robot grid to calculate the neares center. The nearest center is calculated by measuring the euclidean distance to all centerpoints.

def quantityToState(self, position_vector, human):

Takes in a position vector, and outputs the local state. The state is one number, corresponding to the position in the grid array, which is again calculated by measuring the euclidean distance to all centerpoints.

def angleOfStateChange(self):

Makes use of the data which is filtered on either a human or a robot state change. It then calculates the angle between the true positions of the robot when the state changed.

def action(self):

From the angle of state change, the actions for the robot are discretised into six possible actions (this particular discretisation was needed for the experiments). The borders for classifying the action from a certain angle are represented by the black lines in Figure 7. This function also takes into account an extra 'action', that is stagnation.

def globalState(self, localStates):

Given a vector containing a local robot state, a local person state and a person orientation, this function returns the global state of the environment. For performance reasons, the state is calculated only by using linear actions, even though the global state is a position in a 3 dimensional array. Thus, instead, a flattened index is calculated using Equation 3 (in Section 3.2.2, where $N(\mathbf{S})$ is the total amount of states $\mathbf{S}$ in the grid, and θ is the orientation of the person. The function `globalToLocal`, does this the other way around, and determines the original local states from a global state. For completeness, the formulas for achieving this are given in the Equations 4, 5, 6 of Section 3.2.2.

def relativeStateLocation(self, state1, state2, human):

This function takes two neighbouring local states, and tells the position $[0, 5]$ of `state2` relative to `state1` in the hexagonal grid. As previously mentioned, the human is again a boolean which specifies the grid to calculate on. The function returns 6 when `state1` equals `state2`. Similarly, the Python function **def** surroundingStates(self, state, human), can find all the local neighbouring states given a local state. This is particularly useful when

calculating the transitions, since it enhances the performance significantly. When the neighbouring states are known, only those transition functions need to be calculated that are possible in the discretised space (i.e., moving to a neighbouring state).

Calculating the transition probability tables, as described in Section 3.2.2, was implemented in the following way:

```
def probabilities(self, human):
 if human == 0:
  self.robotInActionOut = \
   zip(self.filteredInformation[:-1,1], \
   self.actions, self.filteredInformation[1:,1])
  probabilities = np.zeros([7,7])
  probabilities[6][6]=1
  # It might happen that it is always moving
  for i in self.robotInActionOut:
   sin, action, sout = i
   locations = \
   self.relativeStateLocation(sin, sout, human)
   probabilities[action][locations] += 1
  row = 0
  while row < len(probabilities):
   if probabilities[row].sum() != 0:
    probabilities[row] = \
     probabilities[row]/probabilities[row].sum()
   row +=1
  self.robotProbabilities = probabilities

 if human == 1:
  self.humanInActionOut = \
   zip(self.filteredInformation[:-1,3], \
   self.filteredInformation[:,4], \
    self.filteredInformation[1:,3])
  probabilities = np.zeros([3,7])
  probabilities[0][6] = 1
  # It might happen that it is always moving
  for i in self.humanInActionOut:
   sin, action, sout = i
   locations = \
   self.relativeStateLocation(sin, sout, human)
   probabilities[action][locations] += 1
  row = 0
  while row < len(probabilities):
   if probabilities[row].sum() != 0:
```

```
        probabilities[row] = \
          probabilities[row]/probabilities[row].sum()
      row +=1
      self.humanProbabilities = probabilities
```

The cells of the array contain the probabilities of arriving in the relative next state, given an action. In turn, the transition function can obtain these probabilities, by taking the current local state, the action, and the out state. Note that in the transition function, a change in the person's orientation in the next state should also be taken into account.

```
def globalTransition(self, globalIn, action, globalOut):
```

From the local transition probabilities, the function globalTransition calculates the global transitions. Given the current global state, the action of the robot, and the next global state, it returns the product of the three local transition probabilities. The function `generateGlobalTransitions` iterates over all the possible input states, the possible actions, and the possible output states (as said, only neighbouring states and itself are taken into account), and returns a list of dictionaries. The index of each element in the the list represent a global state and one of the possible actions. Thus, to retrieve the transition probabilities from global state s and action a in the list l, the index that needs to be requested is $a + s \times N(a)$, where $N(a)$ is the total amount of actions. For a certain global state and action, the corresponding dictionary can be consulted, which returns the probabilities of the transitions of getting into a particular next global state (note that this can also be the same state).

To extract data from the human expert, one more function that correlates to the data processing was written during the current project:

```
def experimentsData(self)
```

which uses x, y-pairs from the robot (controlled by the expert), x, y, θ-positions from person, and a target location, to generate a file, again containing a list of dictionaries. Each entry in the list contains a dictionary with the local states, the global state, the action and the target, for each time the global state changed in the data. The expert data is then ready to get processed by the IRL algorithm, which tries to learn the rewards from the expert examples. Using MDPs, subsequently the policies are generated, which can interact with the simulator to acutate the robot.