Activity Report associated to the FCT Grant PTDC/EEA-ACR/73266/2006

A. Reyes and M. Spaan

Abstract

This document sumarizes the activities performed by Alberto Reyes in the FCT Project entitled Decentralized Planning for Multi-Robot Cooperative Systems during his postdoctoral residence in the IS Lab of the Instituto de Sistemas e Robótica (ISR). In this project, we study planning under uncertainty for groups of cooperating multiagent systems. Developing intelligent robots or other realworld systems that plan and perform an assigned task is a major goal of Artificial Intelligence and Robotics. We intend to explicitly consider the communication abilities available to each system and deal with restrictions on available bandwidth or network reliability. In this report, we present: i) a manual model construction and implementation of the Russell's grid problem, ii) an automatic model construction and implementation of a 3–DOF simulated robot (x,y angle), iii) a model specification for a multiple-agent system in the USAR domain, and iv) an implementation of a 2-channel communication module for the Symbolic Perseus planning tool.

1 Introduction

Availability, reliability and quality of communication in real-world multi–robot systems are important factors to decide the use of a centralized vs a decentralized planning model. Allowing agents to communicate with each other increases the team performance, as it mitigates uncertainty: a message from a teammate could provide information about the state of the system, or about the teammate's intentions. In general, a teammate will communicate only if it considers the expected increase in the team performance to outweigh the cost of communication.

In well-structured environments, such as office buildings, one can assume a reliable high-bandwidth communications network, but in unstructured outdoor urban environments communication channels will be less reliable. A particular network topology will also impact the design of the planning algorithm. In a highly structured environment one can assume point-to-point connections among all agents, i.e., the network topology is a fully connected graph. In outdoor environments where robots for instance communicate using wireless communication devices, the graph will not be fully connected. In the case of wireless communication, network connectivity requires a certain signal strength, and hence will be restricted by distance between communication nodes and large obstacles such as buildings. Furthermore, in outdoor scenarios, the robotic agents will be nodes in the ad hoc wireless network themselves. Thus, when deciding where to go, a robot has to consider the impact on the network topology. Another important issue is the question what information should be communicated. Most of the current DEC-POMDP models assume an apriori defined communication language, for instance the observations that each agent receives.

The uncertainty regarding the current and future ability to communicate has to be taken into account in a robot's plan. For instance, it might not be necessary or feasible to remain in communications range at all times, as long as there is a high probability the robots will be to communicate in the near future. Decision making while experiencing uncertainty regarding locations of victims, quality of routes, and communication capabilities are naturally captured in a (decentralized) POMDP model. When designing planning algorithms for these types of problems, centralized or decentralized techniques can be appropriate, depending on the number of robots in a team.

The task of a robot team in the USAR domain is to search for victims in an urban area, for instance after an earthquake. Initially, the location of potential victims is unknown, which leads to uncertainty in a robot's observations: the robot will not be able to tell (with high probability) whether a victim is present at a particular location, until it is close to the location. The robots will be equipped with a highlevel topological map of the area, a graph representation in which nodes can denote locations and the edges are possible routes that a wheeled robot can take. However, when planning a path through the disaster area each robot has to regard the map as uncertain: certain routes between map nodes may have become unavailable due to debris. The quality of a robot's location estimate will vary, as GPS performance in builtup urban areas can be poor. As the robots explore the area, sharing

information regarding located victims or blocked routes will increase team performance. However, the robots will have to cope with a potentially unreliable ad–hoc wireless network. The topology and link quality of the wireless network will change over time, as the robots move through the area.

To apply our techniques, we focused on planning under uncertainty methodology and algorithms for heterogeneous teams of robots involved in indoor search-and-rescue tasks. We also considered scenarios in which several robots have to cooperate in order to locate victims of a natural disaster as fast as possible. In the future, we will build on previous experience with outdoor cooperative navigation of a heterogeneous multirobot team.

This report is structured as follows: In section 2 we show the specification and solution of the grid problem. In section 3 we illustrate a methodology to specify problem in symbolic perseus from machine learning algorithms. A preliminary POMDP model for a multiple-agent system in the USAR domain is explained in section 4. Finally, in section 5 a 2-channel communication module for the Symbolic Perseus planning system is presented.

2 Manual model construction and implementation of the Russell's grid problem

In order to get familiar with the use of planning tools and specially with those focused to the POMDP formalism, we first solved the classic Russell's grid problem [9] using Symbolic Perseus [4]. In the original domain, states are locations in a map represented by a set of cells, however we have changed this representation using nodes $n1, n2, ...n11$ instead of cells. The edges in this new representation denote connectivity paths. Figure 1 shows the original grid problem and its corresponding topological map. The possible noisy actions are discrete orthogonal movements to the right (r), left (l), up (u), down (d), and the null action(). Depending on the agent location, the immediate reward function can assign three possible values: $-1/25, +1, -1$. The motion planning problem is to automatically obtain an optimal policy for an agent to achieve locations with high utility.

This problem was coded and documented in the Perseus's README file to give some directions about the usage of the planning tool. See Appendix A for further details.

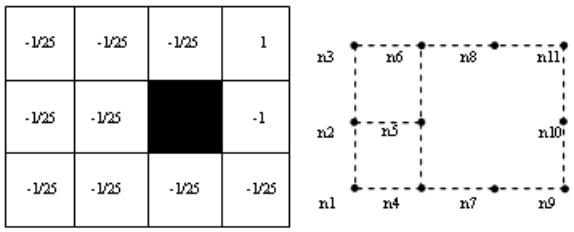


Figure 1: Russell's grid problem (left) and its corresponding topological map (right). Inmmediate reward values are not included in the transformed map.

3 Automatic model construction and implementation of a 3-DOF simulated robot (x,y angle)

We also tested different MDP and machine learning algorithms in a well–structured search domain using a realistic simulator for a single agent with the capability to add white noise to its action execution mechanism. Even when we used a tool for compiling POMDPs, we assumed full observability of states and modeled communication as perfect. In the future, we will consider communication as having a certain cost, with restrictions on available bandwidth and reliability.

3.1 Learning factored models

The MDP model is learned from data based on a random exploration in a simulated environment. We assume that the agent can explore the state space, and that for each state–action cycle it can receive some immediate reward. Based on this random exploration, the reward and transition functions are induced.

Given a set of M non-ordered and rough (discrete and/or continuous) random variables $S^j = X_1, ..., X_n$ defining a deterministic state, an action a^j executed by an agent from a finite set of actions $A = \{a_0, a_1, ...\}$, and a reward (or cost) R^j associated to each state in an instant $j = 1, 2, \ldots, M$, we can learn a factored MDP model as follows:

1. Discretize the continuous attributes from the original sample $D = \{S, R, a\}$. This transformed data set is called the discrete data set $D_d = \{S_d, R_d, a_d\}$. For small state spaces, use conventional statistical discretization techniques. However, in complex state spaces, abstraction techniques are more efficent. For further details see [3, 7].

2. From the subset $\{S_d, R_d\}$ induce a decision tree, RDT, using the algorithm C4.5 [6]. This predicts the reward function R_d in terms of the discrete state variables, $X_1, \ldots, X_n$.

3. Format the discrete data set in such a way that the attributes follow a temporal causal ordering. For example variable $X_{0,t}$ before $X_{0,t+1}$, $X_{1,t}$ before $X_{1,t+1}$, and so on. The whole set of attributes should have the form X_t, X_{t+1}, a_t.

4. Prepare a data set for the induction of a set of 2-stage dynamic Bayesian nets. According to the action space dimension, split the discrete data set into $|A|$ subsets of samples for each action. Remove the attribute a_t from all of them.

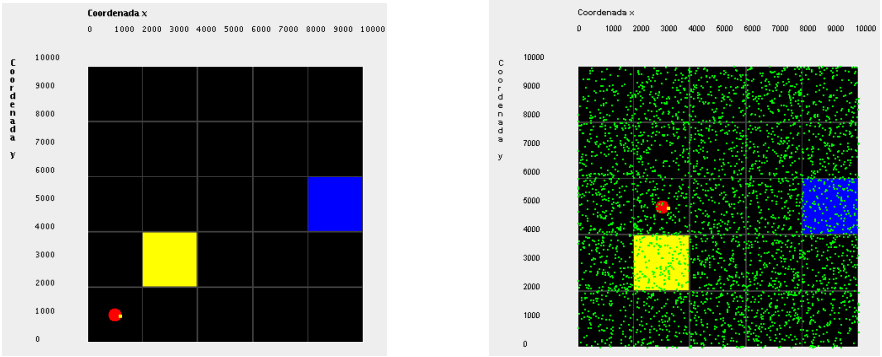
5. Induce a transition model for each subset using the K2 algorithm [2]. The result is a 2-stage dynamic Bayesian net for each action $a \in A$.

This approximate model can be solved using value iteration to obtain the optimal policy. This approach has been successfully applied in other domains [8].

3.2 Experiments

We tested our method in a robot navigation domain using the simulated continuous environment shown in figure 2(a). The complete java-based robot simultor is illustrated in appendix B.1. In this setting goals are represented as dark-color squares with positive immediate reward (300), and non-desirable regions as light-color squares with negative reward (-300). The remaining regions in the navigation area receive 0 reward (black). Rewarded regions are multivalued and the number of rewarded squares is also variable. The robot sensor system included x-y position, angular orientation, and navigation bounds detection. The possible actions in this experiment were: go forward, clockwise rotation (right turn), counterclockwise rotation (left turn), and the null action. In order to simulate real motion, a 10% of Gaussian noise in the action's effect was added.

According to the general algorithm used to learn factored models presented in section 3.1, the first stage to built a decision model is to collect samples from random exploration. Figure 2(b) illustrates the trace of the exploration performed.



(a) Navigation area.

(b) Exploration.

Figure 2: Continuous navigation space and exploration trace.

The reward function mapped from data in a next step is a represented as a decision tree which is illustrated in appendix B.2. There, it can be observed that reward only depends on the x–y position. The reward function was obtained using Weka [10].

Finally, to complete the decision model, the transition function is induced from data. The transition function is represented using a 2-step Bayesian net and it is induced using Elvira [1]. Figure 3 shows the transition function for action $goforward$, and illustrates how if the robot is located at position (s0,s0) with orientation s0 (and it executes action $Goforward$) then, with a joint probability of 0.7 it gets the position (s1,s0) and orientation s0.

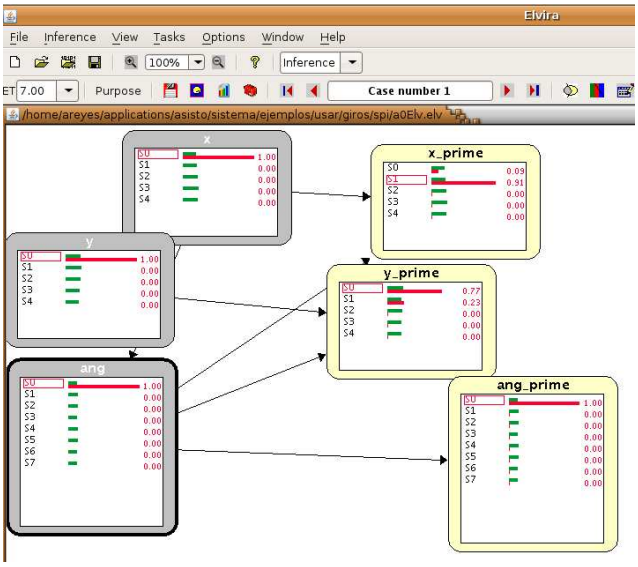


Figure 3: Transition model for the Action 0: Go forward. If the robot is located at position (s0,s0) with orientation s0 then it gets the position (s1,s0) with orientation s0 after executing action 0: Go forward. The joint probability of this transition is 0.7.

3.3 Results and discussion

We solved the problem illustrated in Figure 2(a) using the resulting MDP model aproximated in section 3.2 and value iteration algorithm [5]. With the idea of using a topological representation of the environment in upcoming experiments, we expresed the policy found using a topological map (4). The method successfully guides the robot to move to a likely position with the highest reward. For instance, assuming that the robot has an orientation s2 at the position (s3,s2), the optimal action commands the robot to turn right until it gets orientation s1. In this new state, the robot simply go forward to achieve the goal. The problem with MDP models is that it is not much effective in reasoning about the uncertainty in the belief state, and it will not employ the extended-range sensing action when deemed necessary. MDP-based techniques will not perform this action, as they do not value the quality of information in a sound way.

4 Model specification for a multiple-agent system in the USAR domain

4.1 Task especification

In this section, we describe a problem where a set of robots have to coordinate their actions to do search and rescue tasks during an emergency situation. The test setting is initially inspired in the layout of an IST building.

We consider a scenario in which a robot in cooperation with a number of surveillance cameras has to track and reach a person. The particular scenario is depicted in Figure 5, showing a topological map of the environment and the four locations of the surveillance cameras. For simplicity, we assume we have one robot and one person, but these assumptions can be relaxed. The nodes in the map should be viewed as a coarse discretization of the metric space in which both robot and person operate, suitable for the high-level decision making we consider here. As

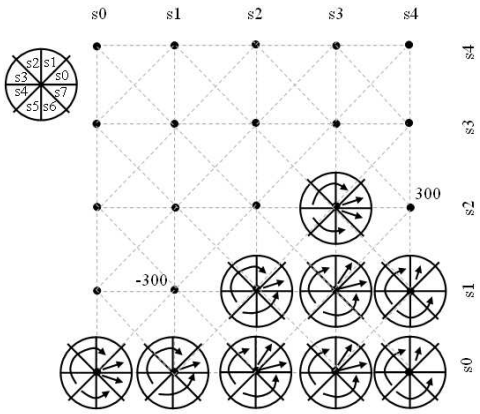


Figure 4: Simplified resulting policy on an equivalent discrete topological map. Assuming that the robot has an orientation s2 at the position (s3,s2), the optimal action commands the robot to turn right until it gets orientation s1. In this new state, the robot simply go forward to achieve the goal.

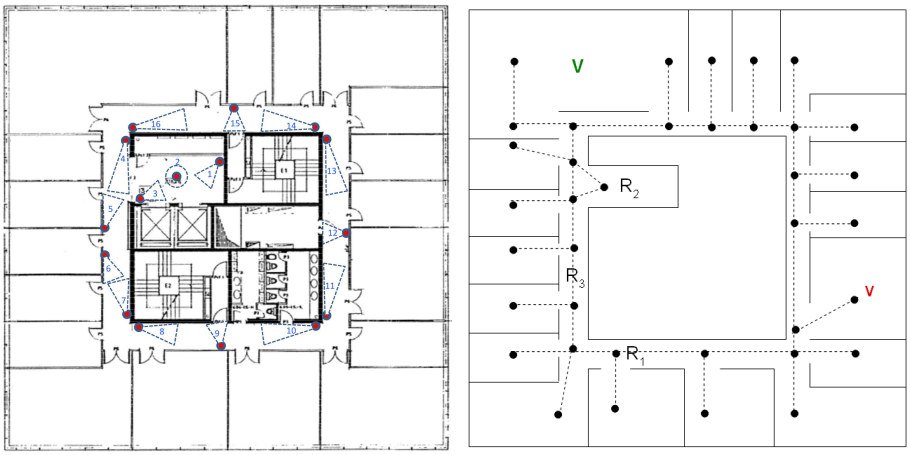


Figure 5: A schematic representation of a floor of ISR/IST's Torre Norte. Shown is a topological map of this environment, represented as a graph. Nodes indicate the possible (discretized) locations of robot and person, and edges indicate the allowed movements between nodes. The locations of the four cameras and their field of view (which node they observe) are indicated as well.

such, we allow person and robot to reside at the same node, since there will be enough space for them to be in the same cell of the environment without colliding.

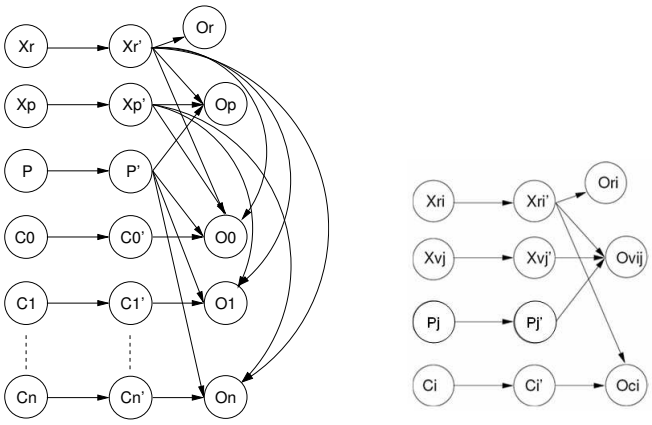
The robot starts in a particular node (top-left one), the person can start in any node with equal probability. As such, the robot has no information regarding the person's location at the beginning of a trial. However, when person is present at the same node as the robot, the robot is likely to detect the person. Also, when the person is in a node observed by a camera, the camera will notify the robot that the person is near it. The robot can also instruct the cameras to switch to a different mode, for instance to run a different detection algorithm that might give less accurate results, but can also detect persons further away (e.g., at neighboring nodes). In general, all these measurements are noisy.

4.2 POMDP model

There are several options to model this scenario as a POMDP, with different features and at different levels of abstraction. Figure 6 shows a POMDP model of the described scenario, formulated as a graphical model. In general, the less variables, and, more importantly, the less connections between them, the more independent all components are and the easier the models are to solve. The person is modeled using its location Xp, i.e., the node of the map at which it resides, and by its properties P. While Xp will change as the person moves through

the environment, P is considered to be static. For instance, the properties might be appearance features like color, which aid robot and cameras to recognize the person. Other properties could include the person's intended destination (e.g., the elevator, a particular office), which will influence the person's behavior. Such a destination might be induced from observations, and can help to guide the robot by predicting better the person's behavior. In this extended model, each of the n cameras C0, C1, ..., Cn is modeled using its own state and each sends an observation to the robot, who has to decide on the course of action. Here we also allow the cameras to observe the person as well as the robot. In this model, the robot has to perform self-localization, for which it can its own observation Or as well as the ones provided by the cameras.

Figure 6(b) shows a reduced POMDP model for the same scenario, which is easier to solve but still captures the essential features of the scenario. Simplifications w.r.t. the extended model are that the robot is assumed to know its position; the person's properties are assumed to be known (i.e., we do not try to induce the person's intended destination); all cameras are considered simultaneously, providing the robot with a single observation. The latter indicates whether the person has been observed near a camera, and if so, which one. The observation is noisy, i.e., when the person is in the same node as a camera (or the robot), there is only a 0.6 probability of detecting it.



(a) Extended POMDP model.

(b) Reduced POMDP model.

Figure 6: An extended and a reduced POMDP model for the scenario we are considering, in which primed variables refer to the next time step. Action links and the reward model have been omitted for clarity. Table 1 provides the legend for the symbols used.

State variables	
Xr	Location of robot: graph node.
Xp	Location of person: graph node.
P	Properties of the person: e.g., appearance features, intended destination.
C0, C1, Cn, Cs	State of each camera: range, detection algorithm used.
Observation variables	
Or	Observation of robot w.r.t. itself, for self-localization.
Op	Observation of person by robot.
O0, O1, On, Oc	Observation made by each camera.

Table 1: Legend of the variables used in our POMDP models (Figure 6).

However, at a higher cost the robot can instruct the cameras to extend their range to include neighboring nodes in the map as well. This means the robot has a field of view (for one time step), but at a lower resolution, as the camera's observation will only tell it that the person is close to it, but not at which node exactly. Note that this is only a simple demonstration of what is possible in this type of POMDP modeling. For instance, we could model the camera as having a field of view of all nodes in a particular corridor, with the level of noise increasing as nodes are further away from the camera. Vision algorithms are more likely to make errors (false positives, false negatives) when operating on fewer pixels, which we can capture naturally in our models.

5 Implementation of a 2-channel communication module for the Symbolic Perseus planning system

To make Symbolic Perseus deal with multi–agent systems more naturally and platform–independent, we implemented a socket-based two-channel multi–agent policy server. This new feature enables different agent implementations for quering policies in real time.

The 2-channel communication module was implemented in Matlab and Java and it is documented in the Perseus' README file. It allows interfacing Symbolic Persues with any code for policy queries purposes. Its main functions are: *tracePOMDPpolicyStationaryServer* and *testpolicyserver*.

tracePOMDPpolicyStationaryServer implements a remote policy server based on the *tracePOMDPpolicyStationary* function and its general form is:

```
tracePOMDPpolicyStationaryServer(host, port,
  pomdpFileName, policyFileName, nSteps,belState).
```

The host argument is the host computer name or ip address where the server is running, and the port argument is the TCP/IP streaming port. pomdpFileName is the problem specification file and the policyFileName is the policy file. As an example for the grid problem, you can simply run:

```
tracePOMDPpolicyStationaryServer('localhost',3015,'data/gridProblem01.txt',
'data/gridProblem01_50bel_10iter_200size.mat').
```

testpolicyserver is a demo implementation of a client for the *tracePOMDPpolicyStationaryServer* function. Its general form is:

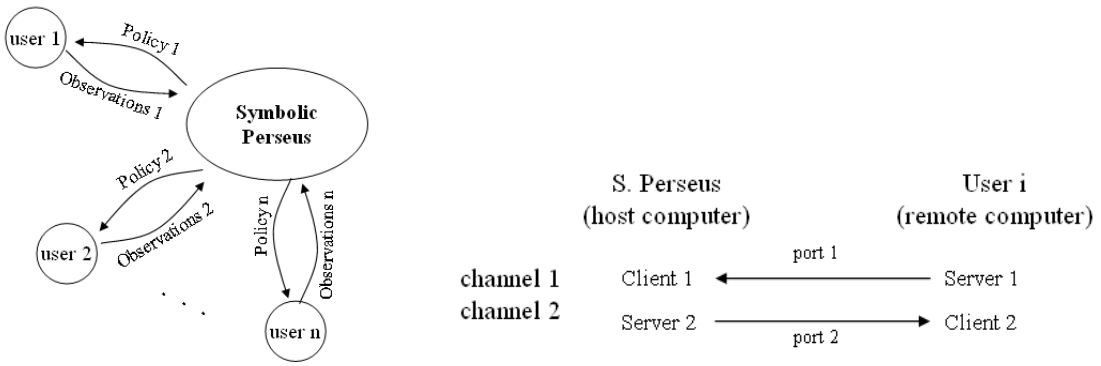
```
testpolicyserver(host, output_port).
```

In this function, the host makes reference to the host computer or ip address where the server is running. The port must be the same one that the tracePOMDPpolicyStationaryServer's. It is also important that the port is not blocked by any firewall. An example to call this function for the grid problem is:

```
testpolicyserver('localhost',3015)
```

.

Figure 7 shows how the policy server function was implemented, and appendix C the resulting code in matlab.



(a) Policy Server.

(b) Client-sever architecture.

Figure 7: Policy server from Perseus

References

[1] Elvira Consortium. Elvira: an environment for creating and using probabilistic graphical models. Technical report, U. de Granada, Spain, 2002.

[2] G. F. Cooper and E. Herskovits. A bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 1992.

[3] Remi Munos and Andrew Moore. Variable resolution discretization for high-accuracy solutions of optimal control problems. In Thomas Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 1348–1355. Morgan Kaufmann Publishers, San Francisco, California, USA, August 1999.

[4] Pascal Poupart. *Exploiting structure to efficiently solve large scale partially observable markov decision processes.* PhD thesis, Toronto, Ont., Canada, Canada, 2005.

[5] M.L. Puterman. *Markov Decision Processes.* Wiley, New York, 1994.

[6] J.R. Quinlan. *C4.5: Programs for machine learning.* Morgan Kaufmann, San Francisco, Calif., USA., 1993.

[7] A. Reyes, L. E. Sucar, E. Morales, and Pablo H. Ibarguengoytia. Abstraction and refinement for solving Markov Decision Processes. In *Workshop on Probabilistic Graphical Models PGM-2006*, pages 263–270, Chezch Republic, 2006.

[8] Alberto Reyes, Matthijs T. J. Spaan, and L. Enrique Sucar. An intelligent assistant for power plants based on factored MDPs. In *Proceedings of the IEEE Int. Conf. on Intelligent System Applications to Power Systems: ISAP 2009*, Curitiba, Brazil, November 2009.

[9] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[10] I.H. Witten. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations, 2nd Ed.* Morgan Kaufmann, USA, 2005.

A Russell's grid problem documentation

A.1 Problem specification using Symbolic-Perseus

```
(variables
(robot node01 node02 node03 node04 node05 node06 node07 node08 node09 node10
node11) // robot location
)
(observations
(robotLoc obsRobot01 obsRobot02 obsRobot03 obsRobot04 obsRobot05 obsRobot06
obsRobot07 obsRobot08 obsRobot09 obsRobot10 obsRobot11) // robot observations
)
init [* (robot
(node01 (0.0))
(node02 (0.1))
(node03 (0.8))
(node04 (0.0))
(node05 (0.0))
(node06 (0.1))
(node07 (0.0))
(node08 (0.0))
(node09 (0.0))
(node10 (0.0))
(node11 (0.0)))
]



dd robotmove1 // nothing
    (robot
        (node01 (robot' (node01 (1.000000))))
        (node02 (robot' (node02 (1.000000))))
        (node03 (robot' (node03 (1.000000))))
        (node04 (robot' (node04 (1.000000))))
        (node05 (robot' (node05 (1.000000))))
        (node06 (robot' (node06 (1.000000))))
```

```
        (node07 (robot' (node07 (1.000000))))
        (node08 (robot' (node08 (1.000000))))
     (node09 (robot' (node09 (1.000000))))
        (node10 (robot' (node10 (1.000000))))
        (node11 (robot' (node11 (1.000000))))
)
enddd

dd robotmove2 // right
    (robot
        (node01 (robot' (node04 (0.800000)) (node05 (0.200000))))
        (node02 (robot' (node04 (0.100000)) (node05 (0.800000)) (node06 (0.100000))))
        (node03 (robot' (node05 (0.200000)) (node06 (0.800000))))
        (node04 (robot' (node07 (1.000000))))
        (node05 (robot' (node05 (1.000000))))
        (node06 (robot' (node08 (1.000000))))
        (node07 (robot' (node09 (0.800000)) (node10 (0.200000))))
        (node08 (robot' (node10 (0.20000)) (node11 (0.80000))))
     (node09 (robot' (node09 (1.000000))))
        (node10 (robot' (node10 (1.000000))))
        (node11 (robot' (node11 (1.000000))))
)
enddd

dd robotmove3 // left
    (robot
        (node01 (robot' (node01 (1.000000))))
        (node02 (robot' (node02 (1.000000))))
        (node03 (robot' (node03 (1.000000))))
        (node04 (robot' (node01 (0.800000)) (node02 (0.200000))))
        (node05 (robot' (node01 (0.100000)) (node02 (0.800000)) (node03 (0.100000))))
        (node06 (robot' (node02 (0.200000)) (node03 (0.800000))))
        (node07 (robot' (node04 (0.800000)) (node05 (0.200000))))
        (node08 (robot' (node05 (0.200000)) (node06 (0.800000))))
     (node09 (robot' (node07 (1.000000))))
        (node10 (robot' (node10 (1.000000))))
        (node11 (robot' (node08 (1.000000))))
)
enddd

dd robotmove4 // up
    (robot
        (node01 (robot' (node01 (1.000000))))
        (node02 (robot' (node01 (0.800000)) (node04 (0.200000))))
        (node03 (robot' (node02 (0.800000)) (node05 (0.200000))))
        (node04 (robot' (node04 (1.000000))))
(node05 (robot' (node01 (0.100000)) (node04 (0.800000)) (node07 (0.100000))))
        (node06 (robot' (node02 (0.200000)) (node05 (0.800000))))
        (node07 (robot' (node07 (1.000000))))
        (node08 (robot' (node08 (1.000000))))
     (node09 (robot' (node09 (1.000000))))
        (node10 (robot' (node07 (0.200000)) (node09 (0.800000))))
        (node11 (robot' (node10 (1.000000))))
)
enddd

dd robotmove5 // down
    (robot
        (node01 (robot' (node02 (0.800000)) (node05 (0.200000))))
        (node02 (robot' (node03 (0.800000)) (node06 (0.200000))))
        (node03 (robot' (node03 (1.000000))))
        (node04 (robot' (node02 (0.200000)) (node05 (0.800000))))
        (node05 (robot' (node03 (0.100000)) (node06 (0.800000)) (node08 (0.100000))))
        (node06 (robot' (node06 (1.000000))))
        (node07 (robot' (node07 (1.000000))))
        (node08 (robot' (node08 (1.000000))))
     (node09 (robot' (node10 (1.000000))))
        (node10 (robot' (node08 (0.200000)) (node11 (0.800000))))
        (node11 (robot' (node11 (1.000000))))
)
enddd

dd perfectRobotLocalization
    (robot'
        (node01 (robotLoc' (obsRobot01 (1.0))))
        (node02 (robotLoc' (obsRobot02 (1.0))))
        (node03 (robotLoc' (obsRobot03 (1.0))))
        (node04 (robotLoc' (obsRobot04 (1.0))))
        (node05 (robotLoc' (obsRobot05 (1.0))))
        (node06 (robotLoc' (obsRobot06 (1.0))))
        (node07 (robotLoc' (obsRobot07 (1.0))))
        (node08 (robotLoc' (obsRobot08 (1.0))))
        (node09 (robotLoc' (obsRobot09 (1.0))))
```

```
        (node10 (robotLoc' (obsRobot10 (1.0))))
        (node11 (robotLoc' (obsRobot11 (1.0))))
)
enddd


action nothing
robot (robotmove1)
        observe
  //robotLoc (noisyRobotLocalization)
  robotLoc(perfectRobotLocalization)
endobserve
endaction

action right
robot (robotmove2)
        observe
  //robotLoc (noisyRobotLocalization)
  robotLoc(perfectRobotLocalization)
endobserve
endaction

action left
robot (robotmove3)
        observe
  //robotLoc (noisyRobotLocalization)
  robotLoc(perfectRobotLocalization)
endobserve
endaction

action up
robot (robotmove4)
        observe
  //robotLoc (noisyRobotLocalization)
  robotLoc(perfectRobotLocalization)
endobserve
endaction

action down
robot (robotmove5)
        observe
  //robotLoc (noisyRobotLocalization)
  robotLoc(perfectRobotLocalization)
endobserve
endaction

reward (robot (node01 (-0.125))
       (node02 (-0.125))
       (node03 (-0.125))
       (node04 (-0.125))
       (node05 (-0.125))
       (node06 (-0.125))
       (node07 (-0.125))
       (node08 (-0.125))
       (node09 (1.0))
       (node10 (-1.0))
       (node11 (-0.125))
)

discount 0.99
tolerance 0.0001
```

B 3-DOF simulated robot problem documentation

B.1 Robot Simulator

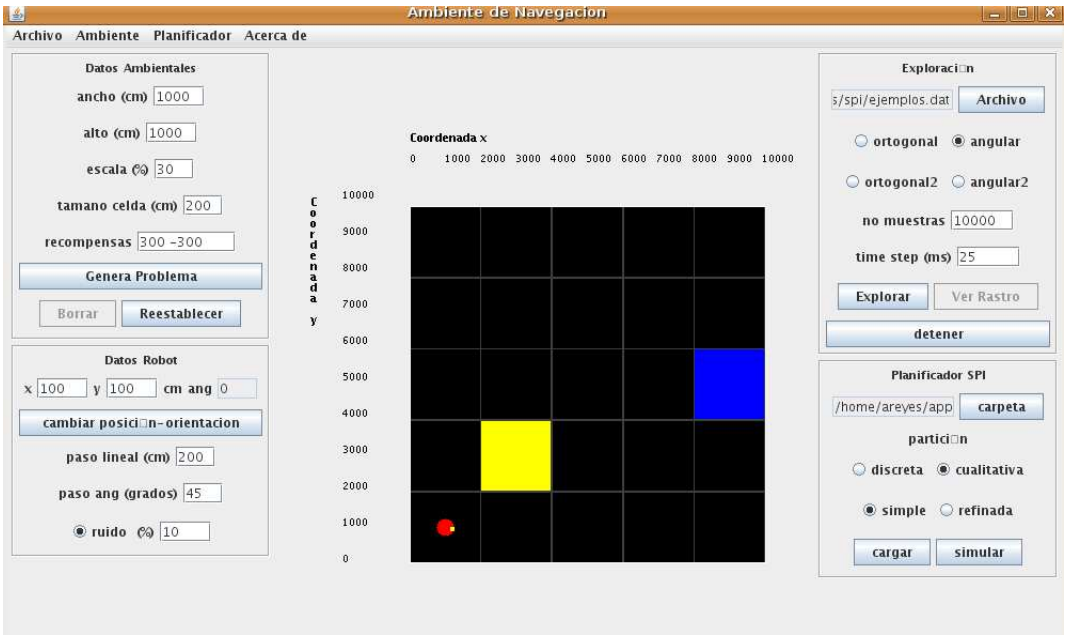


Figure 8: Robot Simulator

B.2 Reward Function

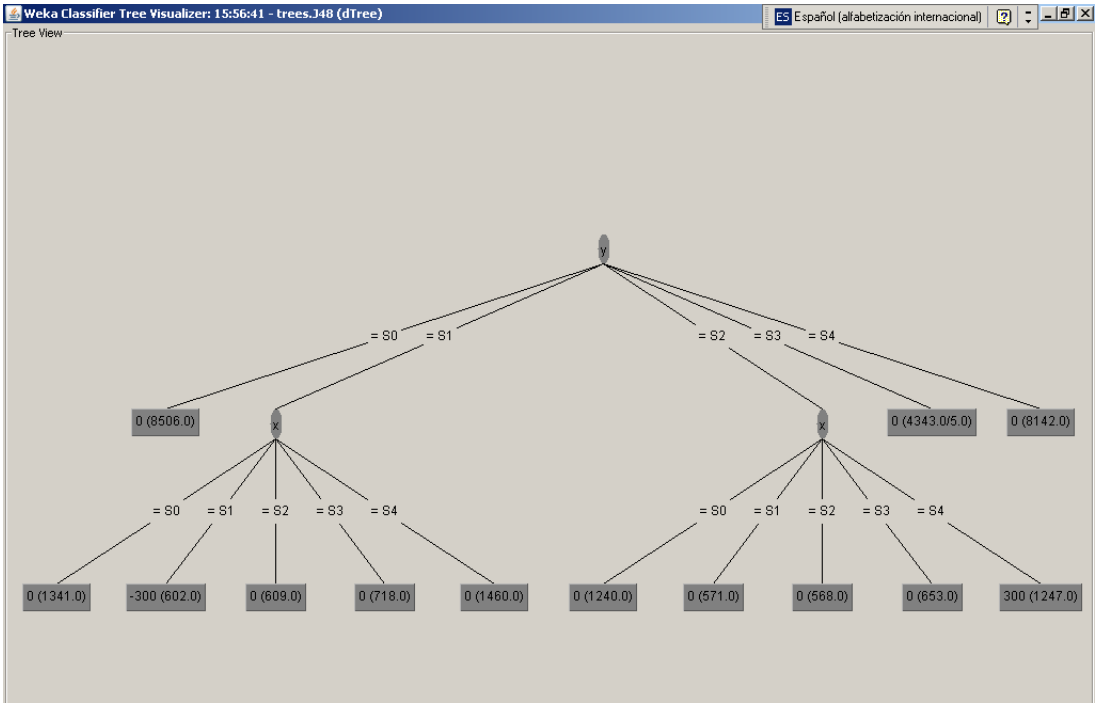


Figure 9: Reward model for the simulated navigation area

C Matlab code for the Symbolic Perseus' Policy Server

C.1 Starting a client

```
% CLIENT connect to a server and read a message
%
% Usage - message = client(host, port, number_of_retries)
function conexion = client(host, port, number_of_retries)

    import java.net.Socket
    import java.io.*

    if (nargin < 3)
        number_of_retries = 20; % set to -1 for infinite
    end

    retry        = 0;
    conexion.input_socket = [];
  %  message      = [];

    while true

        retry = retry + 1;
        if ((number_of_retries > 0) && (retry > number_of_retries))
            fprintf(1, 'Too many retries\n');
            break;
        end

        try
            fprintf(1, 'Retry %d connecting to %s:%d\n', ...
                    retry, host, port);

            % throws if unable to connect
            conexion.input_socket = Socket(host, port);

            % get a buffered data input stream from the socket
            inputStream=conexion.input_socket.getInputStream;
            conexion.inStream=BufferedReader(InputStreamReader(inputStream));
            conexion.dataInStream= DataInputStream(inputStream);

          %  input_stream   = conexion.input_socket.getInputStream;
          %  conexion.d_input_stream = DataInputStream(input_stream);

            % setting output stream PARECE ESTE YA ESTA BIEN
            output_stream=conexion.input_socket.getOutputStream;
            conexion.d_output_stream=DataOutputStream(output_stream);

            fprintf(1, 'Connected to server\n');
%{
            % read data from the socket - wait a short time first
            pause(0.5);
            bytes_available = input_stream.available;
            fprintf(1, 'Reading %d bytes\n', bytes_available);

            message = zeros(1, bytes_available, 'uint8');
            for i = 1:bytes_available
                message(i) = d_input_stream.readByte;
            end

            message = char(message)
      %}
            % cleanup
     %       conexion.input_socket.close;


            break;

        catch
            if ~isempty(conexion.input_socket)
                conexion.input_socket.close;
            end

            % pause before retrying
            pause(1);
        end
    end
end
```

C.2 Starting a server

```
% SERVER Write a message over the specified port
%
% Usage - server(message, output_port, number_of_retries)
```

```
function conexion=server(output_port,number_of_retries)

    import java.net.ServerSocket
    import java.io.*



    if (nargin < 3)
        number_of_retries = 50; % set to -1 for infinite
    end
    retry             = 0;

    conexion.server_socket  = [];
    conexion.output_socket  = [];

    while true

        retry = retry + 1;

        try
            if ((number_of_retries > 0) && (retry > number_of_retries))
                fprintf(1, 'Too many retries\n');
                break;
            end

            fprintf(1, ['Try %d waiting for client to connect to this ' ...
                        'host on port : %d\n'], retry, output_port);

            % wait for 1 second for client to connect server socket
            conexion.server_socket = ServerSocket(output_port);
            conexion.server_socket.setSoTimeout(1000);

            conexion.output_socket = conexion.server_socket.accept;

            fprintf(1, 'Client connected\n');

            % setting output streams

            output_stream   = conexion.output_socket.getOutputStream;
            conexion.d_output_stream = DataOutputStream(output_stream);

            % setting input streams

            inputStream=conexion.output_socket.getInputStream;
            conexion.inStream=BufferedReader(InputStreamReader(inputStream));
            conexion.dataInStream=DataInputStream(inputStream); % bytes


%{
            % output the data over the DataOutputStream
            % Convert to stream of bytes
            fprintf(1, 'Writing %d bytes\n', length(message))
            d_output_stream.writeBytes(char(message));
            d_output_stream.flush;
         %}
            % clean up
        %     conexion.server_socket.close;
        %     conexion.output_socket.close;


            break;

        catch

            if ~isempty(conexion.server_socket)
                conexion.server_socket.close
            end

            if ~isempty(conexion.output_socket)
                conexion.output_socket.close
            end


            % pause before retrying
            pause(1);
        end
    end

end
```

C.3 Send function

```
% sendStr sends a string throughout a socket
%
% Usage - sendStr(message, socket)
function sendStr(message, conexion)

    import java.net.Socket
    import java.net.ServerSocket
    import java.io.*

            d_output_stream = conexion.d_output_stream;

            fprintf(1, 'Writing %d bytes\n', length(message))
            d_output_stream.writeBytes(char(message));
            d_output_stream.write(13);
            d_output_stream.write(10);
            d_output_stream.flush;

end
```

C.4 Receive function

```
% receiveStr receives a string from a socket
%
% Usage - message = receiveStr(socket)
function message = receiveStr(conexion)

    import java.net.ServerSocket
    import java.io.*

        try
            message = [];
            inStream=conexion.inStream;
            message=inStream.readLine;
            fprintf(1, 'Receiving: %s\n',message);

        catch
           % fprintf(1, 'error receiving data\n');

        end

end
```