UNIVERSITY OF MINNESOTA

This is to certify that I have examined this copy of master's thesis by

Shardul Vikram

and have have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Dr. Rich Maclin

Name of Faculty Adviser

Signature of Faculty Advisor

Date

GRADUATE SCHOOL

Acknowledgments

I am grateful to my advisor Dr. Richard Maclin for giving me an opportunity to work under him. I am indebted to him for helping me and for pushing me when I needed it the most. I would also like to thank Dr. Hudson Turner and Dr. Taek Kwon for their cooperation.

Abstract

Numerous methods have been proposed for automatically creating intelligent agents. One promising approach is to create agents that are able to learn based on feedback from their environment. One such technique is reinforcement learning. In reinforcement learning, an agent exists in an environment described by the current state. For example, a robot would operate in an environment where the state is based on the robot's sensors. The agent is then able to choose from amongst a set of actions (e.g., a robot might be able to turn, move forward, extend an arm to grasp an object, etc.) and receives feedback in the form of a reinforcement. A large positive reinforcement would be viewed as a reward for achieving a goal while a negative reinforcement is a punishment. In reinforcement learning, it is often the case that rewards and punishments are sparse (i.e., they are only received when a goal is achieved or a particular bad situation occurs like running into a wall). Thus, an agent cannot learn to simply maximize the reward for its next action. Instead it needs to try to maximize its reward for the *sequence* of future actions it takes. Reinforcement learning provides a mechanism to address this problem.

In this thesis, we investigate a popular form of reinforcement learning called Q-learning. Q-learning has been shown to be effective, but often requires significant training time before achieving good results, thus it may not be suitable for real-world tasks such as robot learning, where training time is expensive. In this work, we investigate a simple learning task involving robotic navigation using Q-learning and demonstrate that the resulting learning is very slow. We will then show that this problem can be somewhat mitigated by using a teacher to bias the robot towards making good decisions during learning. Our tests suggest that while Q-learning will indeed converge towards a solution, it is likely to be too slow for learning environments where training is expensive or slow.

Contents

List of Figures

List of Tables

1 Introduction

The idea of creating intelligent machines has intrigued us for a long time. An understanding of how a machine can be made to learn would open new areas where learning can be applied and provide insight to our own learning abilities. Automatic methods have a long way to go in real situations. They take a long time to learn the task. The learning can however be hastened by using an external teacher. In this thesis, I investigate Reinforcement Learning [Sutton and Barto, 1998] using a real world robot and the method of using a teacher.

1.1 Agent Learning

Learning for an agent is improving performance at some task through experience. Agents can be computer programs or robots that improve from experience. Some of the basic design issues of a learning agent according to [Mitchell, 1997] are:

- *Choosing the training experience* - We can train an agent using different learning methods. The method used affects the learning of the agent. An agent can have examples to learn from. This is *direct learning*. Each example has an input and an output. Another method is *indirect learning* where the agent learns from outcomes of the episodes of performing a task. For a robot, input is the current state of the world and the output is the move it should make. The degree to which the agent controls the learning is another aspect. The agent can rely on a teacher for advice [Lin, 1993, Maclin and Shavlik, I 94], it can learn by making random moves and exploring the environment itself, and it can repeatedly exploit moves it made earlier.

- *Choosing the target function* - In an agent, the target function $\mathcal{V}$ is used to choose the best move from the available ones [Watkins and Dayan, 1992]. For an agent playing chess it chooses a move leading towards a win from among the legal moves. For a robot it is choosing an action from the available ones that moves it towards its goal. In learning problems, an agent is expected to choose an approximation to the target function.

- *Choosing a representation for the target function* - After choosing an approximation to the target function, we need to represent it. There are many options such as a table with states and actions, a collection of rules, a polynomial function or a neural network. In this work, we have represented a problem in the form of a function $\mathcal{V}$. Any example can be represented as a tuple $< p, \mathcal{V}(p) >$, where p is the current state and $\mathcal{V}(p)$ the value of the best move.

- *Choosing a function approximation algorithm* - After the representation of the target function we have to choose an algorithm to find a good function. The algorithm computes the value of the function over a finite number of learning iterations.

For example, a robot uses its sonars to know its state. The sonar returns a vector of continuous values. These values represent the state of the robot. The state is represented by a vector of numbers. We are restricted in using all of the available features of the agent, (e.g., in a robot, the state description is dependent on the values of chosen sensors). In robot navigation, the training experience is a vector of sensor values describing the world, which are used to select a move. The target function is a function that represents the best move from a state. For example, when faced with an obstacle, the best move is to move in a direction around the obstacle.

The target function could be represented as a table of values for states and actions.

1.2 Difficulties in Robot Programming

The learning task I implement is navigating a robot between points in its environment. The robot relies only on the sensor readings to move around by executing a series of actions. The aim is to find the best path to the goal.

One of the difficulties in robot programming is representing the target function and implementing it. The computation needed for calculating the target function should be possible to accomplish within realistic time bounds.

Another difficulty in programming is using the agent's features to interact with the environment. Some properties of a robot's sensors to be taken into consideration while programming a robot are:

- Sensors readings may be uncertain, even in a stable environment.
- The data obtained from the readings is not always an accurate or complete description of the environment.
- Actuators in a robot are prone to errors, and may not produce the desired action.

One example of a possible inaccuracy is that in a dynamically changing environment, sensors can receive signals reflected from other objects in the environment. Sensors often measure aspects of the world indirectly. Some things that sensors may not do are:

- They may not identify objects.
- They do not necessarily give any information about the shape of an object.

- They often do not differentiate between moving and static objects.
- They may be incapable of integrating information obtained from other devices about the environment into one single description.

Even in the case of motor actions, the achieved and desired results differ. Smooth floors, carpets or obstacles in contact with the robot cause odometric errors. Sensing and action rely on the robot's interaction with the environment. To program a robot, its functions and limitations should be known.

1.3 Difficulties with Robotic Learning

Some problems for designing robotic agents that learn include:

- Which algorithm for learning is likely to perform the best?
- How much training is necessary? We have to end the training of the agent after we have a fairly accurate estimation of the target function. However the number of training episodes to get there is unknown.
- When can prior knowledge be used for learning? This could be important if training data is scarce or expensive to obtain.
- What is the best way to reduce the learning task? One way to reduce learning is teaching. If a teacher guides the agent, it need not waste time on random moves for learning a target function.
- How do we represent the environment? We must capture important values about the environment using only sensor readings.

- How can continuous values (sensor readings) be used to represent a state? Continuous values result in a large number of possible states (in theory, an infinite number). A way out of this problem is *discretization* of the continuous values to reduce the number of states.

Different algorithms have been applied to robot navigation, one of them is Q-learning [Watkins and Dayan, 1992]. In Q-learning, the robot executes a series of random actions to change its state and receives a reward for each action executed. The reward function might give a zero reward if the new state is not the goal state and a large positive reward if the new state is the goal state. During learning, actions are taken in a stochastic manner, the robot may take quite some time to reach its goal. Since there is no external teacher to judge the actions of the robot, it will often make exploratory moves, some of which may be bad, like bumping into objects. In contrast, in direct learning, the agent receives feedback about the best action for each state or the value of each action, which allows the agent to converge more quickly.

1.4 Thesis Statement

The purpose of my thesis is to evaluate Reinforcement Learning using a simple real life robot navigation task and determine whether replay (defined below) is effective in learning such a task. In Reinforcement Learning (RL), an agent learns by exploring the environment. In *replay* we repeat a teacher's solution to a task a number of times to make the agent learn it [Lin, 1993]. One question we consider is: in a simple task of teaching a robot to find its way to a position in its environment would RL be efficient in making the robot learn the task within a given time bound? Or is a teacher necessary for learning? To answer this question I first apply Reinforcement

Learning and allow the robot to make random moves to reach the goal. I also use RuleNavigation, a simple navigation program we developed to determine a reasonable number of moves the robot would need to make to reach the goal. Then I used the combined approach of RL using the output of RuleNavigation as a teacher to guide the robot to the goal and find answers to the questions above.

1.5 Layout of Thesis

The next chapter is the background for my thesis. In the background, I describe MagellanPro, the mobile robot used in this thesis. I also describe the Magellan's architecture and the software it runs. There is a brief introduction to sensors and odometry of Magellan, and the problems faced when using them. This is followed by an introduction to Reinforcement Learning. In this I describe the Q-learning algorithm, which is applied to our robot. Reinforcement Learning with a teacher is discussed next. This is followed by a discussion of how to represent the robot's sensors as a Reinforcement Learning problem, in which I describe the representation of the environment using sensor readings.

The third chapter is a description of RuleNavigation, which is our simple method for programming MagellanPro. The chapter explains how an environment description is captured and used later. The problem of obstacle avoidance and reaching a goal is described next. The fourth chapter presents experiments I conducted and results obtained. It looks at the effectiveness of the Reinforcement Learning observing the RuleNavigation. Finally it discusses whether the results of using the RuleNavigation with Reinforcement Learning were any better than that of just using Reinforcement Learning.

The fifth chapter contains conclusions and discusses the thesis, the results and

future work.

2 Background

In this chapter I describe the mobile robot MagellanPro that I used for learning, and the software it uses. I describe Reinforcement Learning and then Q-Learning, a form of Reinforcement Learning (RL), which I applied to programming Magellan. I also present RL with a *teacher* and how Magellan's sensors are phrased as an RL problem. Figure 1 shows the MagellanPro mobile robot used for learning with a mounted camera and radio antennas.

2.1 Description of the Robot

The MagellanPro mobile robot, from Real World Interface, is 16 inches in diameter and 10 inches in height, and weighs 15.9 kilograms. MagellanPro has the following

Figure 1: The MagellanPro from Real World Interfaces Inc. (RWII), with a mounted camera, radio antennas and an on-board PC.

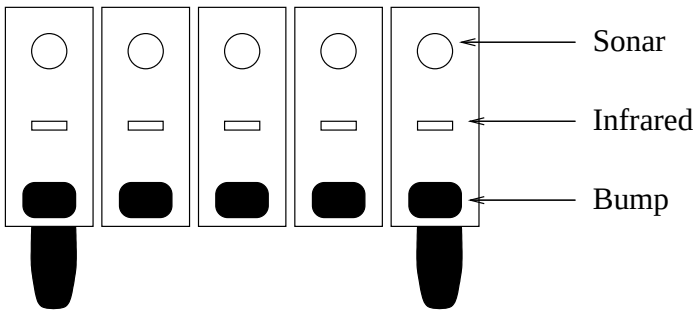


Figure 2: The 3 different types of sensors around MagellanPro (side view). The top sensor is sonar, the middle is infra-red, and the bottom are bump sensors. There are 16 sensors of each kind, with a spacing of around 2 inches

types of sensors: (1) sonar; (2) infrared; and (3) tactile (bump).

The sensors are arrayed around the robot body. There are 16 of each type equally spaced. The sonar sensors are placed around the upper part of the robot, the infrared sensors around the middle part and the tactile sensors around the lower part of the body. Figure 2 is a schematic diagram of the 3 types of sensors of MagellanPro.

MagellanPro has an on-board Pentium based-EBX (Embedded Board eXchange) system [Group, 1997], and communicates with a desktop PC using wireless radio ethernet. An EBX is a single-board computer for embedded applications. The EBX-compliant single-board computer offers functionality equivalent to a complete laptop or desktop PC system. The MagellanPro is assigned an IP address and can be contacted via telnet. It has 2 lead acid batteries which can supply power for 2 to 3 hours. Its turn radius is zero and its drive is 2-wheel differential. It has a maximum translational speed of 1m/second and a rotational speed of 120^o/second.

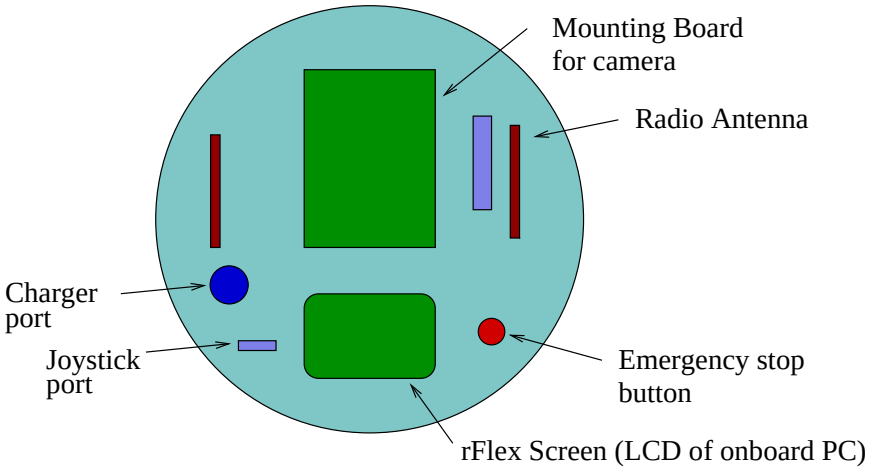


Figure 3: Top view schematic of MagellanPro, showing the rFlex Screen, the two antennas, a stop button to stop the robot in emergency, a joystick and a charger port.

The batteries are replenished using a charger which is connected to the robot for a maximum of four hours with the on-board PC off. If the PC is on, the charger can be connected to the robot for an indefinite period.

The robot can also be driven with a joystick. It has optional accessories which can be mounted on it such as a camera and laser sensors. The MagellanPro we used has a camera mounted on it, but does not have laser sensors. Figure 3 shows the schematic top view of MagellanPro with position of the camera, on-board PC and radio antennas.

2.2 Mobility Robot Integration Software

The MagellanPro uses a software package called *Mobility* [RWII, 1998b]. It is software from RWI introduced in October 1998 for robot development. This software is a distributed, object-oriented toolkit for building control software for single and multiple robots.

Mobility defines the Mobility Robot Object Model (MROM) [RWII, 1998b] using the Common Object Request Broker Architecture (CORBA) 2.X standard Interface Definition Language (IDL) [Michi and Vinoski, 1999]. The MROM defines the robot system as a distributed, hierarchically organized set of objects. Each object is a separate unit of software with an identity, interface and state. Objects represent abstractions of whole robots, sensors, actuators, behaviors, perceptual processes and data storage. Objects provide a flexible model of the robot system that can be reconfigured as new hardware, new algorithms, and new applications are added. Some of the advantages of the Mobility Robot Integration software are:

- Extensibility over time - the existing software can be extended by adding new modules for new hardware components.
- Multiple robot systems - multiple robots can be programmed using Mobility.
- Integration among researchers - a common platform ensures that different researchers can integrate their modules without worrying about portability.

The on-board computer has an LCD screen which is known as the rFLEX screen [RWII, 1998a]. When the computer is powered up a menu appears on the screen, including a user interface to the robot's system such as the sensors, motors and joystick. Text from a remote Linux serial console can be directed to the rFLEX

screen. Mobility addresses a critical issue in robotics research, system integration, where different parts of the robot are accessible through one interface and act as parts of one whole system.

2.3 Mobility Robot Object Model (Language Independent)

The Mobility Robot Object Model (MROM), defines the interfaces and objects needed to represent and manage robot software as a set of concurrently executing, distributed software components. These components represent software abstractions of the robot hardware and behavior. MROM describes a robot as "*a hierarchical collection of object instances that provide interfaces to each component of the robot system*" [RWII, 1998a].

At the top of the MROM hierarchy is a standard CORBA naming service. This naming service allows the software to access many elements of a multi-robot software system. The top level name server contains a directory of robot objects and shared support objects, such as maps shared by a robot team. A robot may include an odometer, tactile sensors, sonar sensors and actuators. All these components are available to the user through a software interface unit called *SystemModuleComponents* [RWII, 1998b]. Figure 4 shows the MROM hierarchy with the naming service at the top and the robot with its subsystems.

2.4 Mobility Object Manager: An Overview

The *Mobility Object Manager* (MOM) is a graphical user interface that lets users observe, tune, configure, and debug the components of Mobility robot control programs as they are running. Figure 5 shows the MagellanPro's sonar and drive view in the MOM. The Mobility Object Manager is the primary tool for interfacing with the

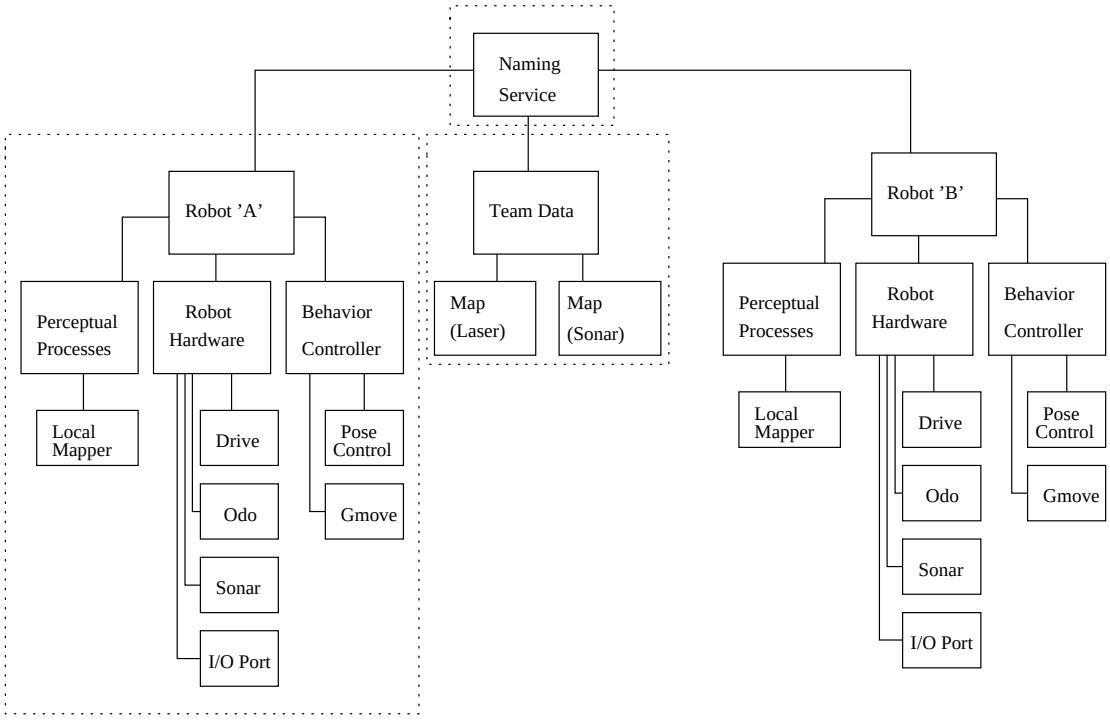


Figure 4: An example of the MROM Setup. Each box represents a mobility component. The MROM is defined using the CORBA Interface Definition Language (IDL) and is based on CORBA object model. Shown here is an example of programming two robots 'A' and 'B' using mobility components. The robots share some common data like maps and use basic mobility components like sonar and drive, to interact and move in an environment.

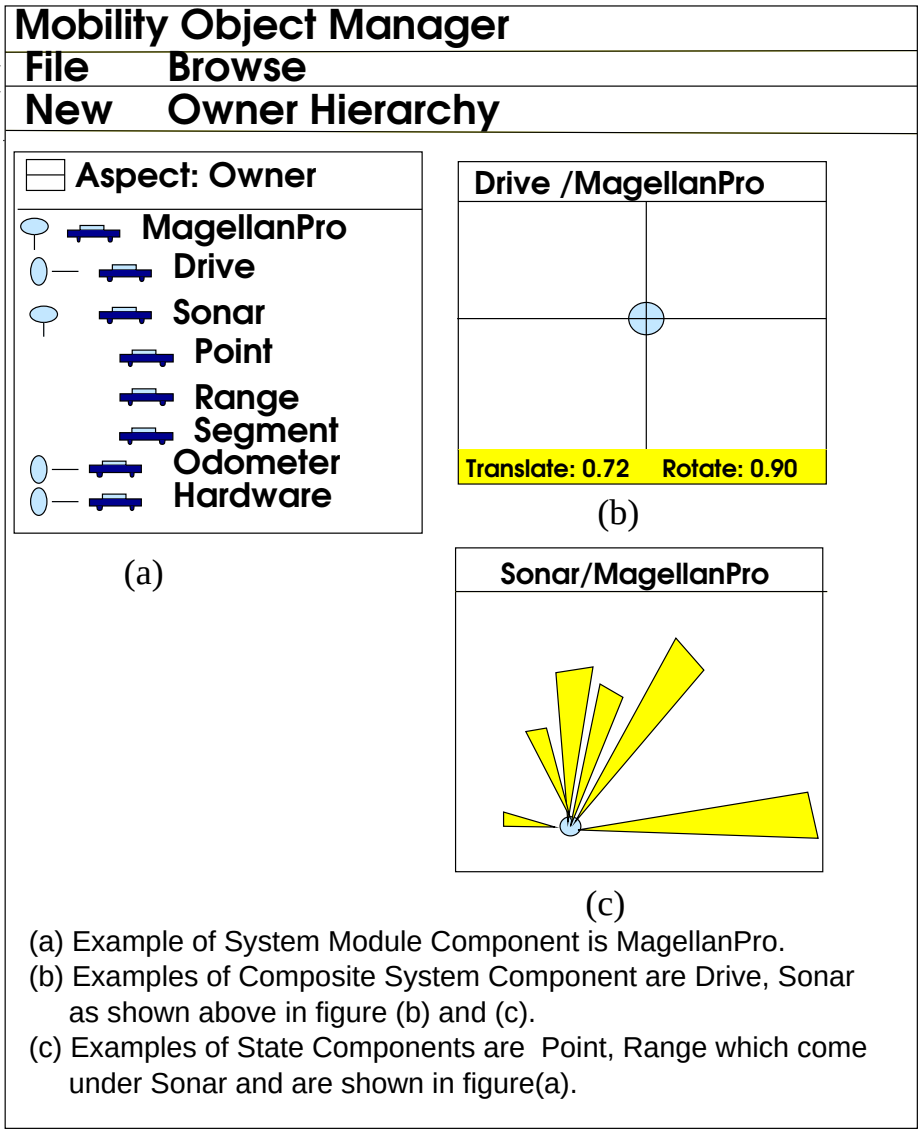


Figure 5: The Mobility Object Manager is a GUI for programming MagellanPro and the various components in Mobility Robot Object Module. Using the MOM, MagellanPro can be driven by dragging the mouse, its sensor readings can be seen and the refresh rates of the sensors can be changed. The Drive shows the translational and rotational velocity of Magellan. The Sonar shows a range view of some of the sensors of Magellan.

robot programs. It is written in Java and communicates with Mobility components through CORBA. It depends on a few background programs running on computers that participate in the Mobility system. A Mobility system is represented by objects that are accessible from a top-level naming service. Because the Mobility Object Manager and Mobility objects controlling the robot are truly distributed object system, there are many ways to arrange a program. The mobility base server must run on the computer physically connected to the robot hardware. Mobility objects may run on the same or other computers. The Mobility Object Manager achieves best performance running directly on the desktop using the *X* windows protocol.

2.5 Basic Robot Components and Interfaces

Some components in Mobility are:

- *SystemComponent* objects - represent robot hardware and software modules.
- *CompositeSystemComponent* - used to aggregate related state and computational objects and treat them as a group. Examples of this are the Actuator system and the Sensor system.
  - *SystemModuleComponent* - is a special CompositeSystemComponent which becomes the 'root' of a collection of objects within a single process.
- *StateComponents* - represent the lowest level of the robot hardware and are updated dynamically and asynchronously. Some basic StateComponents are:
  - *PointState* - is a set of 3D points in space (like sonar target points).
  - *ActuatorState* - represents the actuator status or actuator command value. For example the two velocities of the robot (translational and rotational),

are stored in an actuator state.

– *ImageState* - contains integer image data from the camera.

Mobility's basic robot components provide low-level abstractions of robot-sensors, actuators and physical properties in the form of Mobility System Components. In Mobility, a robot is a hierarchically-arranged collection of elements. The top level of any robot is the *SystemModuleComponent*, a Mobility *SystemComponent* that contains the separate subsystems of an individual robot. Each of these subsystems is itself a *CompositeSystemComponent*. The subsystems contain dynamic *StateObjects*, updated based on the state of the robot sensors and actuators, and properties that allow client programs to discover and adapt to the properties and resources of an individual robot at runtime.

Through the interfaces of these objects, client programs can obtain:

- State information about the robot, where the current position of the robot is relative to its starting position. The starting position of the robot is always taken as the origin.
- The location of robot sensors, the physical location of a sensor on the robot's body. The sensors are numbered zero to fifteen, counter-clockwise.
- Properties of the robot sensor, such as the refresh rate.
- The general shape of the robot body.
- Other basic geometric information determined by the robot hardware.

Clients can also command the robot to move by updating the state of objects contained in the drive object. The basic robot objects, especially sensors, provide

different views of the state information, raw sensor numbers and geometric information like robot-relative target points or contact points.

2.6 Robot Abstractions, Objects and Interfaces

The MROM defines some common abstractions that are useful for building robot control software. These abstractions include representations for: (1) the sensor state; (2) the actuator state; and (3) the physical robot shape.

A sensor is a device which responds to an input quantity by generating a related output, usually in the form of an electrical or optical signal. In Magellan, sonar sensors give the position of an object taking the echo returned from it as input. An actuator is a device that measures a physical quantity and returns its electrical equivalent. The odometer is an example of an actuator, which measures the distance traveled from the rate of revolution of the wheels of the robot. In addition to these fundamental abstractions, there are some basic interfaces for building robot behaviors or control layers for the robot control system.

The concept of state is captured by *StateComponents*. These objects provide simple state buffers that are accessible from anywhere in the Mobility system. This allows *StateComponents* to communicate state changes, and provide access points where the internal operation of the software can be *viewed* remotely for debugging of algorithms.

2.6.1 Sensor Systems

The sensor systems of the robot are each represented as a *CompositeStateComponent* whose properties describe relevant features of a sensor system. A set of *StateComponents* are included as child objects underneath the sensor system components

representing various *views* of the sensor state. These views include: (1) raw range data positions of obstacles in meters; and (2) point positions of obstacles in X,Y coordinates.

2.6.2 How the Sonar Sensors Work

SONAR, an acronym for SOund Navigation And Ranging, models the contours of an environment based on how it reflects sound waves. The sender generates a sonic wave that travels outwards in an expanding cone, and listens for an echo. The characteristics of that echo can help the listener locate objects. MagellanPro reads its sensors three times a second. For each reading, the total time between the generation of the ping and the receipt of the echo, coupled with the speed of sound in the robot's environment, generates an estimate of the distance to the object that bounced back the echo.

As the robot's sonar sensors fire off pings and receive echoes, they continuously update a data structure. Each sonar sensor detects obstacles in a cone-shaped range that covers a half-angle of about 15 degrees and spreads outwards. An obstacle's surface characteristics (smooth or textured), as well as the angle at which an obstacle is placed relative to the robot, significantly affect how and even whether that obstacle will be detected. The sonar sensors can be fooled for number of reasons:

- The sonar sensor has no way of knowing exactly where an obstacle is in its fifteen-degree (and wider) cone of attention.
- The sonar sensor has no way of knowing the relative angle of an obstacle. Obstacles at steep angles might bounce their echoes off in a completely different direction, leaving the sonar sensor ignorant of their existence, as it never

receives the echo.

- The sonar sensor can be fooled if its ping bounces off an obliquely-angled object onto another object in the environment, which then, in turn returns an echo to the sonar sensor, but at a seemingly further distance. Figure 6 shows how edges of a sonar beam cause error when reflected off a surface.

- Extremely smooth walls presented at steep angles, and glass walls, can seriously mislead the sonar sensors, because of their ability to reflect sound. Figure 7 shows that a sonar beam can be reflected off a surface in such a way that it does not return to the sensor.

To combat the problem, the robot has multiple sonar sensors, providing redundancy and enabling cross checking. Sonar sensors almost never underestimate the distance to an obstacle. The reason for this is that an edge of the cone may be reflected off the object back to the sensor. As a result the distance measured would be less than in the case where the object was lying in the cone of reflection. Figure 6 illustrates this point. Thus it is a good idea to examine the distances returned by a group of sensors and use the lowest value.

2.6.3 Odometry and Position Control

The robot's mobile base is equipped with wheel encoders that keep track of the revolutions of the wheel as the robot travels about its environment. The robot's motion controller integrates these measurements to estimate the robot's current position at any time with respect to its original position; that is, where it was when it started rolling. While this measurement is highly accurate for short distances, errors can and do accumulate as the robot travels further afield. By itself, the

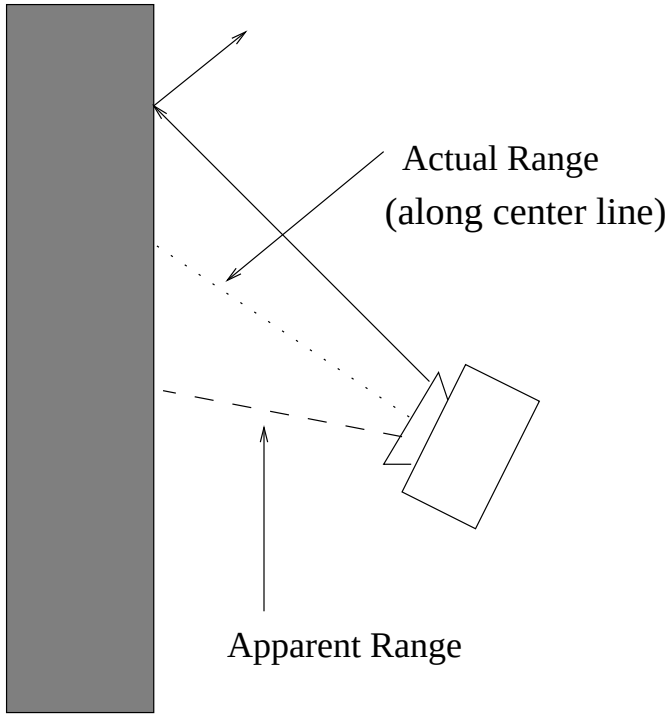


Figure 6: Range error caused by an edge of beam reflection. In the figure it is shown that one edge, the left one, of the cone is reflected back from the wall while the other (right one) is not. The actual distance is the measured by the reflection of the waves along the centerline of the cone. Because of the way the waves are reflected back a range error occurs.

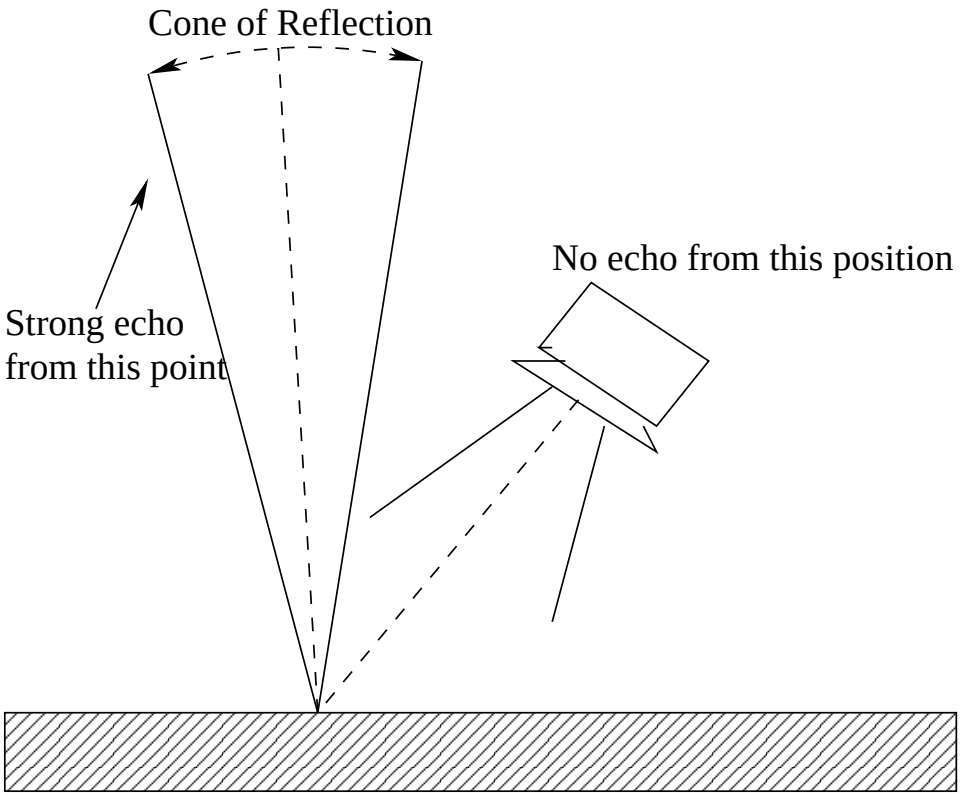


Figure 7: How sonar can be fooled: angular errors. The cone hits the wall at an angle. Following the laws of reflection it is reflected away from the robot sensor. A strong echo occurs when an object is perpendicular to the cone of reflection, which is not the case here because the cone has bounced off the object. In such cases the robot sensor may receive an echo that has been reflected many times or it may receive no echo at all. In each case the actual distance of the object cannot be determined.

robot's motion controller hardware has no way to detect wheel skid or errors in wheel tracking, routine hazards in real-world research and operational environments plagued with slippery floors, carpeting and doorjambs.

Mobility provides odometry information computed from robot wheel rotation. The robot position and velocity can be determined in X, Y distances relative to the start-up location. The odometry output is in meters and radians and the velocity output is *meters/second* and *radians/second*.

2.7 Reinforcement Learning

In Reinforcement Learning (RL) [Sutton and Barto, 1998], an autonomous agent senses and acts in an environment and learns to choose actions to achieve its goals.

For example, a robot is given a task of moving from one end of the room to the other. For simplicity we assume that the room is divided to form a grid. The robot moves from one cell of the grid to the other. The chairs and tables in the room are obstacles in the robot's path and they occupy a number of cells in the grid. A robot can move from one empty cell to another but not into one which is already occupied by objects. The robot moves around these obstacles into empty cells and should eventually reach the goal. Since there can be more than one way to move to the other end of the room, one question is which path should the robot take? The answer is the best path out of the available ones to reach the goal avoiding obstacles in the way. Therefore the task of the robot is to learn to make its way to the goal following a good path. A *control policy*, which the robot learns, determines the path to the goal. A *policy* π is a function $\pi(s) \rightarrow a$ that maps states to actions, that is for each state it indicates the action the agent should take. An action is a move which changes the agent's present state and results in a new state. How does the

robot follow the control policy? The robot observes its sensor readings to select an action. In each state the control policy, learned by the robot, indicates the action to take (e.g., to move to the left, to the right, forward or backward). The action taken in a state results in a new state. For each action the robot takes it receives a reward. For example, the reward for reaching the goal could be a large positive value, for running into an obstacle a large negative value, and for other moves a zero. Following the control policy, the robot attempts to choose actions which maximize the accumulated reward in each state.

The policy evaluation in RL differs from other *function approximation* problems because it takes the following into consideration:

- Delayed Rewards - In each state the robot performs an action and receives a reward. We might think that we have a fixed table of $<s,a,r>$, where for an action a in state s the robot receives a reward r. This is however not the case. Assignment of reward r for an action a in state s may be delayed until a far distant goal is achieved.

- Exploration/Exploitation tradeoff - In RL, the agent makes moves probabilistically during learning. By making moves it has not tried before it *explores* yet unknown state-action combinations. It can also repeat the set of moves which maximized its rewards earlier, this is *exploitation*. The agent faces a tradeoff in choosing *exploration*, finding new states to maximize rewards, or *exploitation*, following the policy which has yielded high rewards in the past.

- Partially Observable States - For a robot moving forward, the description of the environment is given by the sensors in the front. This may provide only partial information about the environment because only a subset of the total

set of sensors are read.

- Multitask Learning - Robots usually do not learn a single task. The actual task may contain subtasks which should be learned first. The robot learns one subtask and then moves on to learn the next. For example, navigating through a corridor while picking up a tennis ball from the floor. This consists of two subtasks: navigating and picking up a tennis ball. The robot should be able to use knowledge of previous moves in learning a new task so that the time to learn a new task is reduced.

2.7.1 Some Examples of Reinforcement Learning

RL examples are common in our life, from learning to drive a car to baking a cake. Consider driving a car, which is initially difficult. We need to keep the car steady by turning the steering to the left or the right. If the car moves more towards the left, we steer it right, if it moves towards the right we steer it left. As time passes we improve and are able to keep the car steady without moving the steering too much.

Following are some more examples of RL:

- In chess a move is determined by the state of the board. If the next state is a winning state, a high positive reward is given. If it is a losing state a high negative is given. For other states the reward is zero.

- Pole balancing on a cart [Chambers and Michie, 1968], is a classic planning problem. To balance the pole, force is applied in the direction opposite to which it is falling. The force depends on the position of the falling pole. The force is small if the pole is almost straight and large if it is far from the center.

The reward is a high negative if the pole falls and zero if it does not.

- A robotic arm learns to pick up a can of juice. It searches the space for the can. For picking up the can it receives a large positive reward. For knocking the can down it receives a large negative reward. For moves resulting in neither of the two it receives a zero reward. The robot can be given small rewards for completing subtasks. For example, moving near the can results in a small positive reinforcement. For aimlessly searching in space a small negative reinforcement might be given.

2.7.2 Reinforcement Learning Environment

A state is said to have the Markov Property if we are able to determine the next state by considering *only* the current state and the action taken from the state. An RL task that satisfies the Markov property can be modeled as a Markov Decision Process or MDP [Mitchell, 1997, Sutton and Barto, 1998].

In RL, an agent has a set of states S and a set of actions A it can perform. At any given time t, the agent can be in any one of the states. From that state it can execute one action out of the available set of actions A. For performing an action it gets a reinforcement (a reward) r that represents feedback about taking that action in that state. This interaction between the agent and its environment is shown in Figure 8. The action puts the agent into a successive state where it selects a new action. The task of the agent is to learn a policy which can help it select the best action in each state. The cumulative reward is calculated by adding the rewards that would be received in the future, by following the policy, discounted by some factor. If we follow the policy π from some state s_t, the cumulative reward we would get is:

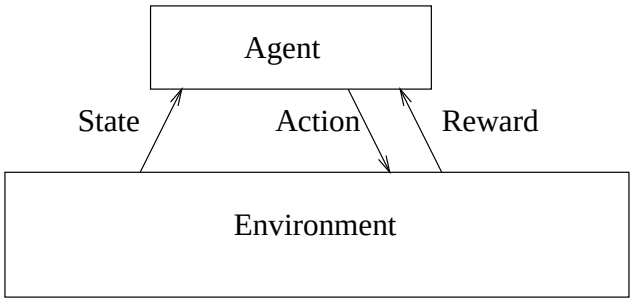


Figure 8: The agent-environment interaction. The agent perceives its position in the environment, the *state*, executes an *action*, and receives a *reward* in return for the action. For example, a state could be the position of the robot with respect to an obstacle and the set of possible actions would be moving left, right or back. The reward would be -10 if it hits the obstacle and +1 for avoiding it.

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \quad (1)$$

$$= \sum_{i=0}^{\infty} \gamma^i r_{t+i} \quad (2)$$

where r_{t+i} is the reward received by taking an action when in state s_{t+i}. From state s_t, the agent moves to state s_{t+1} and receives reward r_t. From state s_{t+1} it moves to s_{t+2} and receives a reward r_{t+1} discounted by a factor γ. All of this while the agent follows the policy π. The constant γ, $0 \leq \gamma < 1$, determines the relative value of delayed versus immediate rewards. Rewards received i time steps into the future are discounted by a factor of γ^i. If $\gamma = 0$, only the immediate rewards are considered. As γ is set closer to 1, future rewards are given greater emphasis relative to immediate rewards. We do not use $\gamma = 1$ because in most cases we prefer to obtain the reward sooner rather than later. The quantity $V^{\pi}(s)$ is the *discounted*

future reward achieved by following the policy π from the initial state s.

$$\pi^* = argmax V^\pi(s), \forall s \tag{3}$$

The states, actions and reinforcements in a problem can be illustrated by taking the robot MagellanPro as an example. Assume the task for the robot is reaching a goal state which lies at coordinates (X,Y) relative to its present position. The values of the sensors in Magellan represent a state. The actions possible are turning to the left, turning to the right and moving forward. Choosing actions is not a simple task because of the range of available angles in which the robot can turn. For example, turning to the left can be turning by anywhere from say 37^o to 58^o. We therefore allow only a small finite set of discrete actions. For instance turning to left could be our action number 1, and the robot will turn by a fixed angle 30^o. Each act of turning to the left or the right is followed by moving a small distance in that direction. Each action executed by Magellan brings it into a new state. It receives a reward of -1 for entering a state which is not the goal. For bumping into objects it receives a high negative reinforcement, say -100, and for reaching the goal a high positive reinforcement, say +100. The rewards for each action are decided by the trainer.

An RL agent follows the general plan shown in Table 1. It maintains a table of all possible states and its current estimate of $V^\pi(s_t)$ values. This value is the amount of reward the agent would recieve if it starts from state s and follows the policy π. There are several actions possible from a state. Each time an action is taken in a state the agent attempts to update the $V^\pi(s_t)$ value. The policy of the agent would

Table 1: Following is the general plan followed by an agent. The discount factor γ is a constant $\in [0, 1)$.

General Reinforcement Learning

- For each s_t initialize the table entry $V^{\pi}(s_t)$ to a small random value.
- Observe the state s.
- Do forever:
  - Select an action a and execute it
  - Receive immediate reward r
  - Observe new state s_{t+1}
  - Update table entry for $V^{\pi}(s_t)$ using the equation:
    $V^{\pi}(s_t) \leftarrow r + \gamma V^{\pi}(s_{t+1})$

---

be to take an action to maximize the $V^{\pi}(s_t)$ of a state s or the cumulative reward it receives when starting from state s. If a new policy gives higher cumulative rewards than the present one then the agent follows the new one. This is policy improvement and with improvement the agent moves towards learning the best policy, one which will give the highest possible cumulative reward $V^{\pi}(s_t)$ for a state s.

2.8 Reinforcement Learning and Dynamic Programming

Dynamic Programming (DP) solves a problem by finding solutions to sub-problems and combining them to get a final solution. DP is often applied to *optimization problems*, where many solutions are possible for the same problem.

DP takes the following into consideration when finding the best policy: (1) policy evaluation; (2) policy improvement; (3) policy iteration; and (4) value iteration.

2.8.1 Policy Evaluation

Policy evaluation for a problem is computation of the cumulative rewards obtained by an agent over a period of time by following a policy π. It is likely that this is difficult or intractable to directly calculate for any real world problem.

2.8.2 Policy Improvement

The value function is used to find a good solution to the problem. There may be a policy that is better than our current one. If there is such a policy then we should follow that one instead of our current one. How do we know that a policy is better? The answer is by comparing the cumulative rewards obtained by following both policies we can decide whether we need to change to a different one or not. Therefore if by following a policy π' we get a higher cumulative reward $V^{\pi'}$ than by following π, we should follow π'. This helps in finding the best policy which is the one that gives the highest cumulative rewards.

2.8.3 Policy Iteration

A better policy is found by policy evaluation followed by policy improvement. A policy is first evaluated, its value function is calculated, then a better policy is found if it exists. If an improved policy exists we follow the new policy. Thus the best policy can be found by repeatedly doing policy evaluation and policy improvement until no further improvements can be made. This way of finding the best policy is known as policy iteration.

2.8.4 Value Iteration

Policy evaluation may involve heavy computation, a drawback to the policy iteration method to find a better policy. Another method used to find the better policy or a better value function is *value iteration*. Value iteration finds the best policy in a number of iterations. The terminating condition of the value iteration algorithm is when there is very little change in the value function (V^{π}) in two successive iterations. For each state we have an estimate of V^{π}, the total cumulative amount received if starting from that state. Each action taken stochastically changes the policy being followed and changes the cumulative reward received in that state. By doing *exploration* and *exploitation* we improve the policy.

All RL algorithms try to achieve the same results as Dynamic Programming (DP) algorithms. One of the reasons DP algorithms are not suited for large problems is the amount of computation involved in finding the best policy.

2.9 Q Learning

Q-learning [Watkins and Dayan, 1992] maps state-action pairs to values (Q-values). We make use of a function called the Q-function to find the best policy instead of the value function V. Each action in a state has a Q-value. The Q-value of an action is the sum of the discounted reinforcement received when that action is executed and the current policy is followed. The value of the state is the maximum Q-value in that state.

Q-learning extends the value iteration of DP. Q-learning can be applied to deterministic and non-deterministic Markov Decision Processes (MDPs). In a deterministic MDP, an action in a state s_t always results in the same state s_{t+1}. In a non-deterministic MDP the same actions may result in a set of states s_{t+1} with

various probabilities.

Not all actions in a state are performed to calculate the cumulative rewards resulting from successor states. Only the action with the maximum probability is chosen. When an action is taken, its Q-value is updated by adding the reward received upon executing that action and the discounted maximum Q-value of the resulting state.

2.9.1 Q-Function

The Q-function updates the Q-values of the action taken in the current state. The Q-value, Q(s,a), of an action is the reward received immediately upon executing action from the state, plus the value (discounted by γ) of following the best policy thereafter:

$$Q(s,a) = r(s,a) + \gamma max_{a'} Q(\delta(s,a),a) \quad (4)$$

$$Q(s,a) = r(s,a) + \gamma max_{a'} (Q(s',a)) \quad (5)$$

where the function that determines the new state in which the agent should move into by taking action a in state s is $\delta(s,a)$ and the reward received by taking that action is determined by the function $r(s,a)$. The new state resulting from taking action a is *s'*. The Q(s,a) value for a given state s and an action a from that state is the cumulative reward the agent would receive starting from state s and reaching the goal. The Q(s,a) values for each state and action are stored in a 2-dimensional table containing entries for all possible state/action pairs. This entry is the Q(s,a) value of action a in state s. Each Q(s,a) value for an action is updated following the above given rule when that action is executed. We choose the maximum *Q(s,a)* in order to choose the best action a in state s:

$$\pi(s) \quad = \quad argmax_a Q(s,a) \tag{6}$$

This is the agents policy. The value of Q for the current state and action summarizes in a single number all of the information needed to determine the discounted cumulative reward that will be gained in the future if action a is selected in state s. Figure 9 shows a simple maze world where Q-learning is applied to find an optimized path to the goal along with the cumulative rewards in each state and the Q values for each action from a state.

Earlier we considered Q-learning in a deterministic environment. In a non-deterministic case, the reward and the action executed at some state are probabilistically chosen. In robot problems with noisy sensors and effectors it is appropriate to consider actions and rewards as nondeterministic. Here the rewards and the actions leading to new states are viewed as producing a probability distribution over outcomes based on the states and the actions, and then choosing the action with maximum future reward. When these probability distributions do not depend on previous states and actions, the system is called a non-deterministic MDP. In the nondeterministic case we take into account that the outcome of actions are no longer deterministic. We redefine V^π of a policy π to be the *expected value* (over these non-deterministic outcomes) of the discounted cumulative reward received by applying this policy

$$V^\pi(s_t) \equiv E\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right]$$

To define the algorithm we use $\widehat{Q}$ to refer to the learner's current estimate of the actual Q-function. The following rule updates the $\widehat{Q}$ values by keeping a count

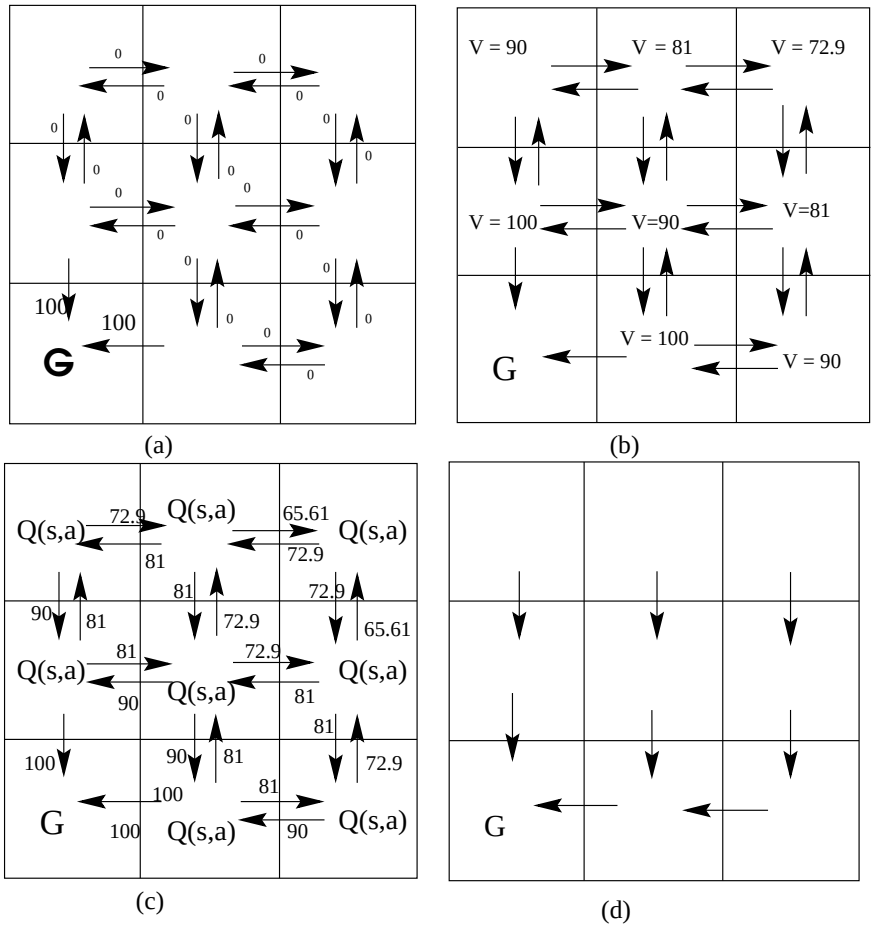


Figure 9: A maze world is a simple example of Reinforcement Learning. Part (a) shows possible moves from one state to the next. For each action leading to a state which is not the goal state, the reward received is 0. The maximum reward for moving into the goal state is 100. The value V in part (b) is the sum of discounted future rewards the agent will receive when it starts from that state. The best policy might yield the shortest path to the goal. One such policy is shown in part (d). Q(s,a), the values on each arrow in part (c), are the maximum discounted cumulative reward that can be received starting from s and taking an action a.

of the number of times a state is visited. The value of $\widehat{Q}$ is updated by taking a weighted average over the number of visits to the state:

$$\widehat{Q}_n(s,a) \quad = \quad (1-\alpha_n)\widehat{Q}_{n-1}(s,a) + \alpha_n[r(s,a) + \gamma max_{a'}\widehat{Q}_{n-1}(s',a')] \qquad (7)$$

where

$$\alpha_n \quad = \quad \frac{1}{1+visits_n(s,a)} \qquad (8)$$

s and a are the state and action updated during the nth iteration and $visits_n(s,a)$ is the total number of times this state-action pair has been tried including the nth iteration. The new state in which the agent is thrown into is *s'* and the action in this state which yields the highest cumulative reward is *a'*.

2.9.2 An Algorithm for Q-learning

The key problem is finding a reliable way to estimate training values for Q, given only a sequence of immediate rewards r spread over time. This can be accomplished through iterative approximation.

$$V^(s) \quad = \quad max_{a'}\widehat{Q}(s,a') \qquad (9)$$

$$\widehat{Q}(s,a) \quad = \quad r(s,a) + \gamma max_{a'}\widehat{Q}(\delta(s,a),a') \qquad (10)$$

In this algorithm (see Table 2) the learner represents its hypothesis $\widehat{Q}$ by a large table with a separate entry for each state-action pair. The table entry for each pair *(s,a)* stores the current estimate for $\widehat{Q}(s,a)$, the learner's hypothesis of the actual but unknown value *Q(s,a)*. The table is filled initially with small random values. The agent repeatedly observes the current state s, chooses some action a,

Table 2: The Q-learning algorithm for deterministic rewards and actions. The discount factor γ is a constant $\in [0, 1)$.

Q learning algorithm

- For each s,a initialize the table entry $\widehat{Q}(s, a)$ to a small random value.
- Observe the state s.
- Do forever:
  - Select an action a and execute it
    - Initially choose actions randomly
    - As number of iterations choose actions with highest $\widehat{Q}$
    - Assign a probability to each action using the formula $\frac{1}{1+e^{-\widehat{Q}(s,a)}}$
    - Select an action stochastically
    - Execute that action
  - Receive immediate reward r
  - Observe new state s'
  - Update table entry for $\widehat{Q}(s, a)$ using the equation:
    $\widehat{Q}(s, a) \leftarrow \widehat{Q}(s, a) + \mu((r + \gamma max_{a'} \widehat{Q}(s', a')) - \widehat{Q}(s, a))$
  - $s \leftarrow s'$

---

executes this action, then observes the resulting reward $r = r(s,a)$ and the new state $s' = \delta(s, a)$. It then updates the table entry for $\widehat{Q}(s, a)$ following each transition with the following rule:

$$\widehat{Q}(s,a) \leftarrow \widehat{Q}(s,a) + \mu((r + \gamma max_{a'}\widehat{Q}(s',a')) - \widehat{Q}(s,a)) \tag{11}$$

where μ is the *learning rate* parameter.

This training rule uses the agent's current $\widehat{Q}$ values for the new state *s'* to refine its estimate of $\widehat{Q}(s,a)$ for the previous state *s*. Each action is assigned a probability between 0 and 1 by using the function $\frac{1}{1+e^{-\widehat{Q}(s,a)}}$, using their $\widehat{Q}$ values. The action with the highest probability is the action which returns the maximum cumulative rewards. The agent executes an action in its new environment and then observes the resulting state *s'* and the reward *r*. It can be viewed as sampling these functions at the current values of *s* and *a*. Using this algorithm the agent's $\widehat{Q}$ estimate converges in the limit to the optimal *Q*-function, provided the system can be modeled as a nondeterministic MDP, the reward function *r* is bounded, and actions are chosen so that every state-action pair is visited infinitely often [Mitchell, 1997]. Once the optimal policy is reached, *Q(s,a)* directly approximates the optimal action-value function.

2.10 Reinforcement Learning with a Teacher

Reinforcement learning for adaptive control generally involves less human interaction than many other learning techniques. The problem with RL methods is that the convergence is slow. They also assume a MDP, where only the current state determines the next state, which does not hold for many robot problems. An agent can be made to deal with a range of non-Markovian tasks by relying on prior knowledge [Lin, 1993, Maclin and Shavlik, I 94].

2.10.1 Experience Replay and Teaching

One approach to speeding up RL is to use a teacher. The teacher demonstrates how the goal can be reached from some initial states by providing a lesson [Lin, 1993]. Each lesson contains the record of the states, actions and rewards encountered during the teacher's problem solution.

In the case of MagellanPro, our RuleNavigation program acts as the teacher. In RuleNavigation, written directly using Mobility, Magellan is given a goal state and it moves towards it recording the states it encounters on its way towards the goal. The states which have been recorded become a lesson.

A lesson is a record of the series of states and actions encountered by the agent in the past. For example, the information at time t of the robot being in state s_t and taking an action a which results in state s_{t+1} with reinforcement r_t is kept as a tuple during learning and is known as *experience*. The sequence of experiences collected in a learning trial is called a *lesson*. After the trial the *lesson* is *replayed* by the learning algorithm to update the initial $\widehat{Q}$ function. The agent, in this way, can save many action executions required to learn a good control policy. The collection of experiences is in fact a model, used by the agent, representing the environment's input-output patterns.

Teaching is useful in three ways:

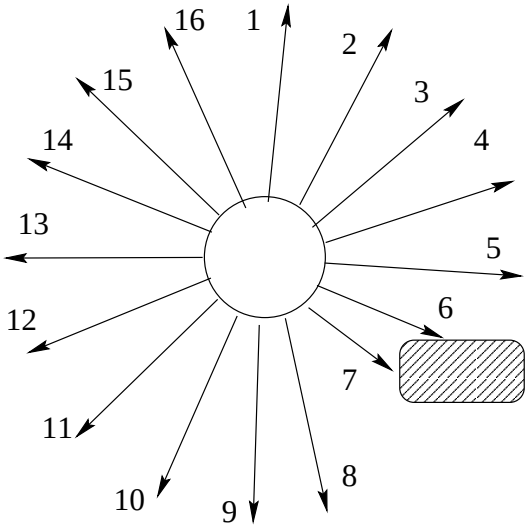
- Partial solutions can be readily made available from the teacher.
- The learner is directed by teaching to first explore the parts of search space which possibly contain the goal states. This is important when the search space is large and a thorough search is impractical.
- The learner can avoid getting stuck in a local minima during the search for the

best policy by relying on teaching. The reason for this is that the learner is dependent on the teacher, the teacher knows where the goal is and the learner would be pulled out of local minima when it takes the teacher's advice.

3 Robot's Sensors Phrased as a Reinforcement Learning Problem

The sensors in MagellanPro have a range of four meters. A sensor returns the location of an object in X, Y coordinates. The distance is measured in meters. Figure 10 shows the readings of the sonar sensors of Magellan when an object is nearby. A problem in applying RL to a robot in a real world environment is determining the states of the robot at each time step. The actuators and the sensors operate using continuous values and therefore the states and actions are described in real values. This is a problem for standard Reinforcement Learning, which assumes discrete states and actions. To take each real value as a state would be impractical, resulting in a large Q-value table.

To address this problem we divide the continuous values in ranges and represent the sensor readings in these ranges. Thus two sensor readings sr_1 and sr_2 would have the same value if they lie in the same range. For example, if sensor 10 returns 0.51 meters for an object and sensor 9 return 0.40 meters both these readings will have value 3 because they lie in the same range, which is 3. The range distances used in training Magellan are shown in Figure 11. By converting the continuous values of the sensor readings into discrete ones we are ready to apply standard RL. We have also reduced the size of the Q-value table considerably by using discrete values. The size of the Q-table is dependent upon the number of sensor values we use to represent a single state. The number of actions possible from any state are three: left, right and front. Left and right actions result in 30 degree turns. Only three actions were taken to restrict the number of actions available to the robot, which affects (reduces) the size of the Q-table.



1	4.00
2	4.00
3	4.00
4	4.00
5	4.00
6	1.345
7	1.246
8	4.00
9	4.00
10	4.00
11	4.00
12	4.00
13	4.00
14	4.00
15	4.00
16	4.00

Figure 10: The sensor readings when an object is near the robot. The position of an object is returned in X, Y coordinates. The distance of the object from the robot, where the robot is considered the origin, is $\sqrt{X^2 + Y^2}$. The distances shown in the table above are the Euclidean distances calculated from the readings of each sensor.

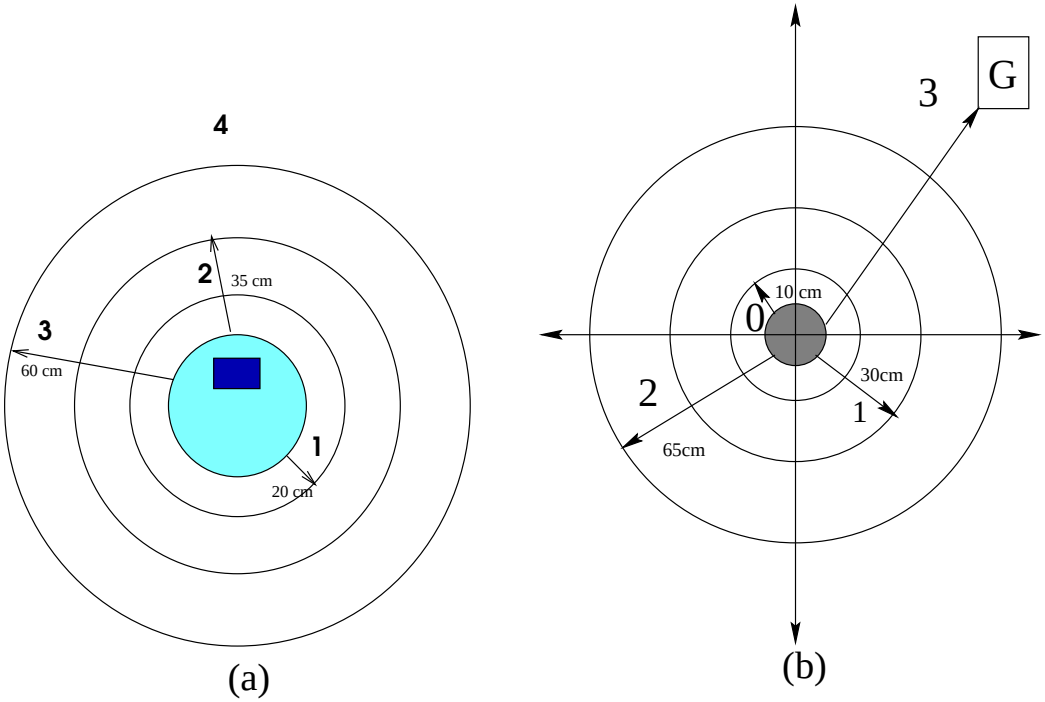


Figure 11: The distance discretized by dividing it into ranges. The sensor readings are then represented in these values instead of continuous values. This reduces the number of states in which a robot can be. The number of states possible are 4^n, where n is the number of sensors used for representing a state. The distance from the goal is also discretized. The distance is represented in the coordinate form as X and Y distance from the goal. The Y distance is positive if the goal is to the left of the robot and negative if it's on the right. The X distance is positive if the goal lies ahead of the robot and negative if it is behind it.

In the case of MagellanPro, the sensor readings are first scaled to continuous values between 0 and 1. The distance of four meters is scaled to 1 as it is the maximum distance the sensors can read. Any object lying beyond four meters has a reading of four meters on the sensor irrespective of its position. However this maximum distance is not included in the view of the robot. Anything around the robot within a radius of 60 centimeters makes up the environment of the robot. The distance of perception is kept small because it is easier to navigate and avoid obstacles, and the closer the object the more sensors will detect it. Therefore, by considering the reading of a group of sensors, an obstacle can be located with some precision.

The sensor reading is divided into $four$ ranges. Nine sensor readings are used to represent a state. Figure 12 shows this vector with an object lying to the left of it. The readings represent the surroundings of the robot. These nine values are calculated by first taking readings in three different views of the environment and then overlapping the resulting values to produce a single view. The three views are the sensor readings of the view in front of the robot, the view at angle θ to the left of the robot and the view at an angle θ to the right of the robot. The final vector has 7 sensor values and two values to represent its current position from the goal in X and Y coordinates. If goal is within 10 centimeters then range is 0, for a difference that lies between 10 and 30 centimeters the range is 1 and for a difference from 30 to 65 centimeters the range is 2. Beyond 65 centimeters the range is 3. The range for Y is positive if robot is to the left of the goal and negative if it is to the right. Similarly for X coordinate the range is positive if the goal is ahead of the robot and negative if it is behind the robot. If the robot's present coordinate are (1,0) and the goal is at (1,-1) then its position is (0,-4). The distance of the robot from the goal

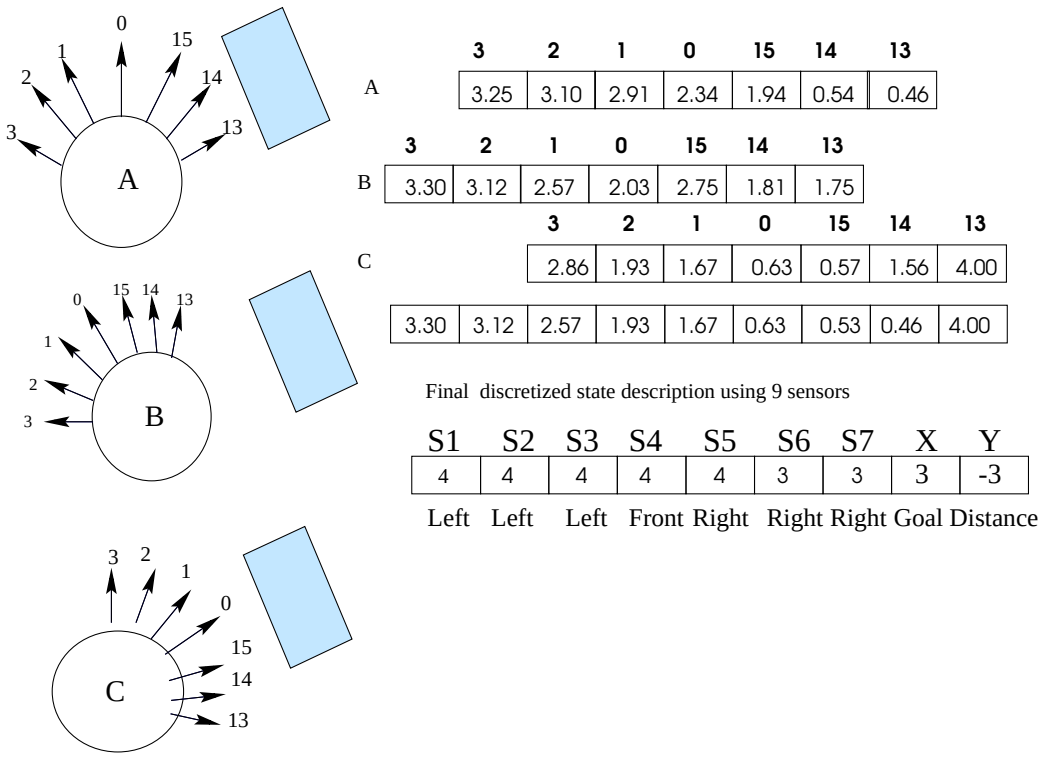


Figure 12: The state is represented by nine sensors. The values of the sensors are calculated by overlapping the sensor readings in 3 different directions. The continuous values are then discretized to form the needed state description. The numbers 0,1,2,3,4,13,14,15 are the sensors that are read. The creation of the sensor array can be compared to taking a scenic picture but in three parts and then combining them to form a single view. Such a picture would have some overlapping areas which need to be combined to form one scene. The same is the case here where three different readings are taken and combined into one. In case of overlapping sensors the minimum of sensor readings is considered. Out of the nine sensor readings the two extreme readings are discarded. These 7 sensor readings with the distance of the robot along X,Y axis from the goal, form the final state description of 9 values. Here we have assumed that the goal is to the front and right of robot at a distance greater than 60 cm.

along the X axis is 0 meter which falls in range 0 and the distance along Y axis is 1 meter to the right of the robot which falls in range -4 and therefore the position (0,-4).

After obtaining the final view of the surroundings, the available actions (actions are turning to the left, turning to the right and moving straight) are defined. This is done by reading the range values, which give an estimate of the object distance from the current position of the robot. Each action is then given a random initial value between 1 and -1, which is the Q-value for that action in a particular state. The size of the table formed was $4^7 \times 9^2$.

The values considered for discretization for sensor readings as shown in Figure 11 (a) were:

- Value 1: Range is less than 20 cm.
- Value 2: Range is between 20 cm and 35 cm.
- Value 3: Range is between 35 cm and 60 cm.
- Value 4: Range is greater than 60 cm.

The thresholds for X and Y distances from the goal as shown in Figure 11 (b) were:

- Value 0: If goal lies within a distance of 10 cm.
- Value 1: If the goal lies within a distance of 10 cm to 30 cm.
- Value 2: If the goal lies within a distance of 30 cm to 65 cm.
- value 3: If the goal lies beyond 65 cm.

When the RL program begins it asks for the position of the goal. It then defines its initial state. Based on the table created internally it makes a stochastic move, it moves a short distance about 20 cm in the chosen direction. After it makes a move it updates the table and chooses an action again. For each move, it receives a reward of -1 if the new state is not the goal state. The robot uses bump sensors to see if it has hit an obstacle or not. If it collides with an object it receives a high negative reward of -200. For moving into the goal state it receives a positive reward of +200. The reason for giving a reward of -1 for states that were not goal states is that we want the robot to take actions it did not take before and explore the environment. With newer actions it may find policies better than the one it currently follows and change to a newer one.

To test RL with a teacher, the RuleNavigation program is run again and each state the robot enters is recorded. The RuleNavigation program is a simple navigation program. The robot calculates the direction of the goal and moves towards it. After each move it stops to capture the state description and the action taken to move out of the current state. The nine valued state description along with the action taken is stored in a file. This is done until the robot reaches the goal. The stored state descriptions and actions are a lesson. The lesson is replayed offline multiple times to update the agent's $\widehat{Q}$ estimate and acts as a guide to the robot when RL is applied again. The Q table is not randomized initially but the one which replay was performed is used.

4 The Task

A basic task in programming robots is navigation. Figure 13 shows a sample navigation task. Many algorithms and methods have been applied to the problem of robot navigation through an obstacle filled environment [Choset and Burdick, 1995, Wolfram et al., 1999], to improve performance.

The problem is teaching a robot to navigate around objects and to reach a goal. The goal can be as simple as reaching a corner on the other end of the room or as complex as docking to a battery charger in the other room. The robot is dependent only on its sensors for moving in an environment. The performance of the robot depends on its internal description of the environment. Since the sensors are not always accurate, we must expect an approximate description of the state instead of a perfect one. As the robot moves it creates a table of sensor readings and based upon them it decides where there is space available.

4.1 Basic Robot Program to Solve the Problem

The basic robot program, RuleNavigation, does not use RL. The robot creates and reads an array of sensor readings to determine its moves. The robot's sonar sensors and odometry are the only source of information about the environment and about itself. The odometer readings give the position of the robot relative to the starting position. The position is given in X, Y coordinates and direction θ of heading. The state description of the environment is created only when the robot nears an obstacle. The continuous values of the sensors pose a problem for defining the states of the robot. In the real world we deal with continuous values and the number of states in which the robot can be is very large. To deal with this problem the readings are

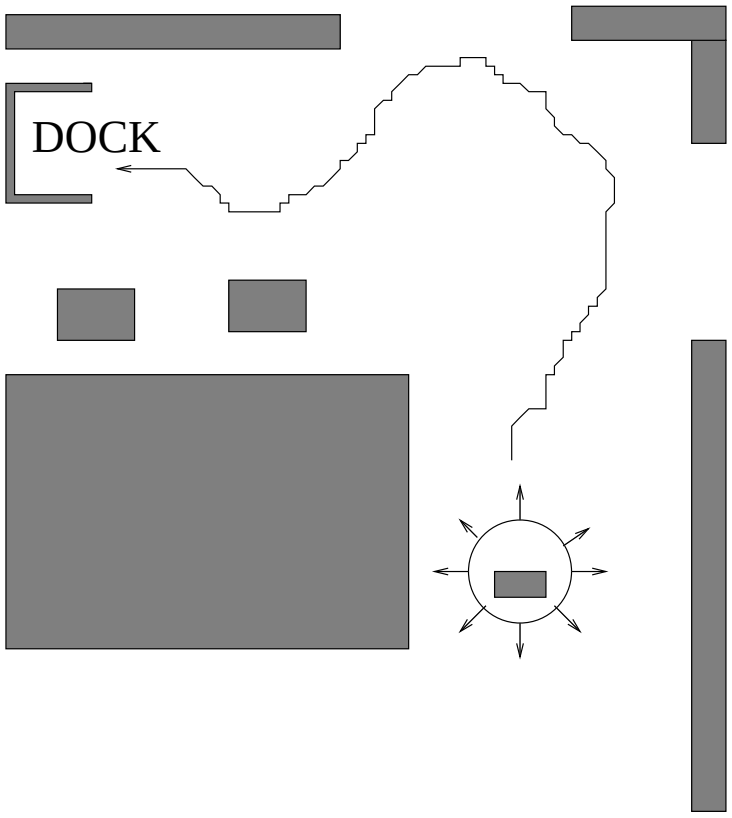


Figure 13: Navigation is a basic task in robotics. A robot navigates through an environment with obstacles and makes its way towards the goal, here the DOCK. The task may also be navigating through a number of rooms searching for battery re-charger. Advanced internal map creation methods [Thrun and Buecken, 1996], are used for such complex tasks, to guide the robot with accuracy around obstacles.

discretized, which reduces the number of states, which can still be large depending upon the number of sensors used to get the description of the environment. The state description used in navigating is an array of nine sensor values. This state description is created by overlapping readings from the same sensors when they face the left, right and the front.

We have written a simple navigation program to move between two points in the robot's environment, avoiding any obstacles in between. Our program, RuleNavigation runs by taking the coordinates of the destination as input. It calculates the direction in which the goal lies and moves in that direction. The sensor readings are constantly updated to check for any approaching obstacle. If an obstacle comes within a desired range, the robot stops, creates a table of sensor readings, and finds available moves to avoid the obstacle. The robot turns in that direction and moves a small distance. It then checks the goal's position relative to its own and corrects itself by turning in the direction of the goal and moving towards it. For example if the destination coordinates were given as (1,1) and the robot was at its starting position, the direction to move would be 45^o (calculated as $arctan\frac{End_y - Origin_y}{End_x - Origin_x}$) to the left, based on its coordinate system where the axis have been changed, the X-axis in the real world being the Y-axis in the robot's system and vice-versa. If an obstacle appears in the middle, the robot updates its sensor readings and creates the nine sensor value table which for example may have a value [4,4,4,3,1,1,2,2,2] for the nine sensors, then the robot after reading the table would place its position in the front and to the right. This is calculated by taking the minimum of the three rightmost readings [1,2,2] and of the three in the middle [3,1,1] of the table. The available space is on the left [4,4,3], where the robot will move.

RuleNavigation is very basic. It works by creating a table of the sensor readings

and reading it to make a move. The state description is created only when the robot is near an obstacle. There is no knowledge of previous states encountered. The navigation is guided by the direction in which the goal lies.

There are some errors involved in running RuleNavigation. The errors are due to the robot's own systems like sonars and odometer. It can happen that the robot bumps into objects because the sonar is late in detecting it. This happens quite often. The actuator system of the robot introduces errors in the motion of the robot. As the robot moves, small errors due to the actuator system accumulate over time and cause the values to be inaccurate. Therefore the achieved and the desired direction of motion are not the same. This is a problem when precision is required of the robot.

5 Experiments

We conducted five experiments where Magellan Pro was trained to navigate using RuleNavigation, Reinforcement Learning (RL) and RL with RuleNavigation as a teacher. The task was to move between two points, about 1.1414 meters apart. The starting point was fixed as was the origin. The end point lay to the right and front of the robot. In the robot's coordinate system it was the point (1,-1). In RuleNavigation the robot would turn in the direction of the goal and move towards it. In the case of the other experiments, RL and RL with a teacher, the robot made stochastically chosen moves. In the act of choosing an action it might collide with walls or wander away from the goal. If a collision occurred we would start again, otherwise we let the robot explore the environment. In RL with a teacher the RuleNavigation was run and all the states encountered by the robot and the actions taken from those states while moving towards the goal were recorded. The start and the end points were the same in all experiments. The reason for choosing such a small distance was to avoid errors due to the actuators to affect the results.

5.1 Effectiveness of RuleNavigation

RuleNavigation was run 5 times in each experiment. The program ended successfully with the robot reaching the goal. When there is no obstacle the robot turns and moves towards the goal. When an obstacle lies in the path, the robot finds a way to avoid the obstacle and then moves on to find the goal. Figure 14 shows how RuleNavigation moves the robot from its starting position to the goal. RuleNavigation takes into account dynamic objects. The robot stops and recreates the sensor table of the environment, if a dynamic object passes close to it. The time taken to

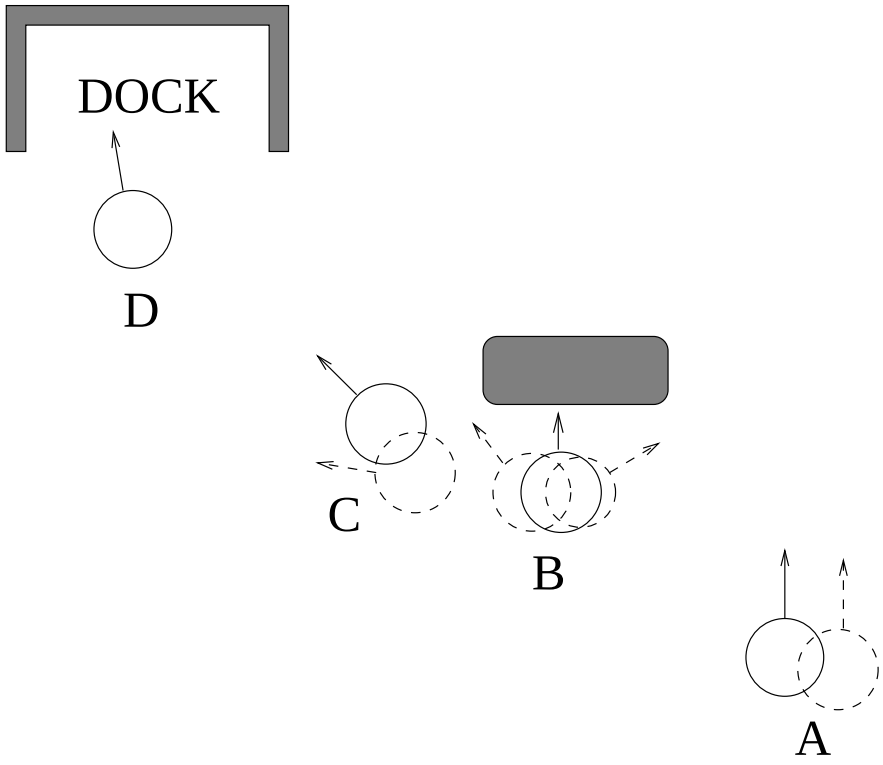


Figure 14: Rule Navigation approach to obstacle avoidance. The robot turns and moves in the direction of the goal, from its initial position A. If it encounters an obstacle at B, it creates an internal table of the environment (state description) and moves to C, avoiding the obstacle. At C, it recalculates the direction of the goal and moves towards it. D is the final position, the goal, for the robot to reach.

reach the goal without any obstacle in the way, depends on the speed of the robot. I used a translational velocity of 0.15 m/second. The goal position is given in the coordinates, taking the robot's position as the origin.

5.2 Reinforcement Learning Results

During learning, a reward of -1 was given to a move leading the robot into a state which is not the goal, a negative reward of -200 for colliding with objects and +200 for reaching the goal. Figure 15 shows how the robot will navigate if it learns using RL.

The robot had a training phase of 10 episodes and then an evaluation phase of 5 episodes. In the training phase the robot was made to learn the task. It made stochastic moves to reach the goal based on the current predicted Q-values of the possible moves. The probability of making a random move was high initially and then decreased slowly as the training continued and the robot picked the action with the maximum Q-value later. In the evaluation phase the robot just chooses the move with the highest Q-value. In both the phases the starting position of the robot was the same.

The total number of training phases in each experiment were 200, divided into 20 training phases of 10 episodes each. The number of evaluation phases were 20, one for each training phase, of 5 episodes each. The robot was stopped and made to start all over again if a collision occurred or if it reached the goal in the training phase. In the evaluation phase the number of steps before it collided or it took to reach the goal were noted. The average time for a training phase with 10 runs varied from an hour and a half to two hours. The average time for evaluation was forty five minutes to an hour and fifteen minutes. We did not set a limit to the number

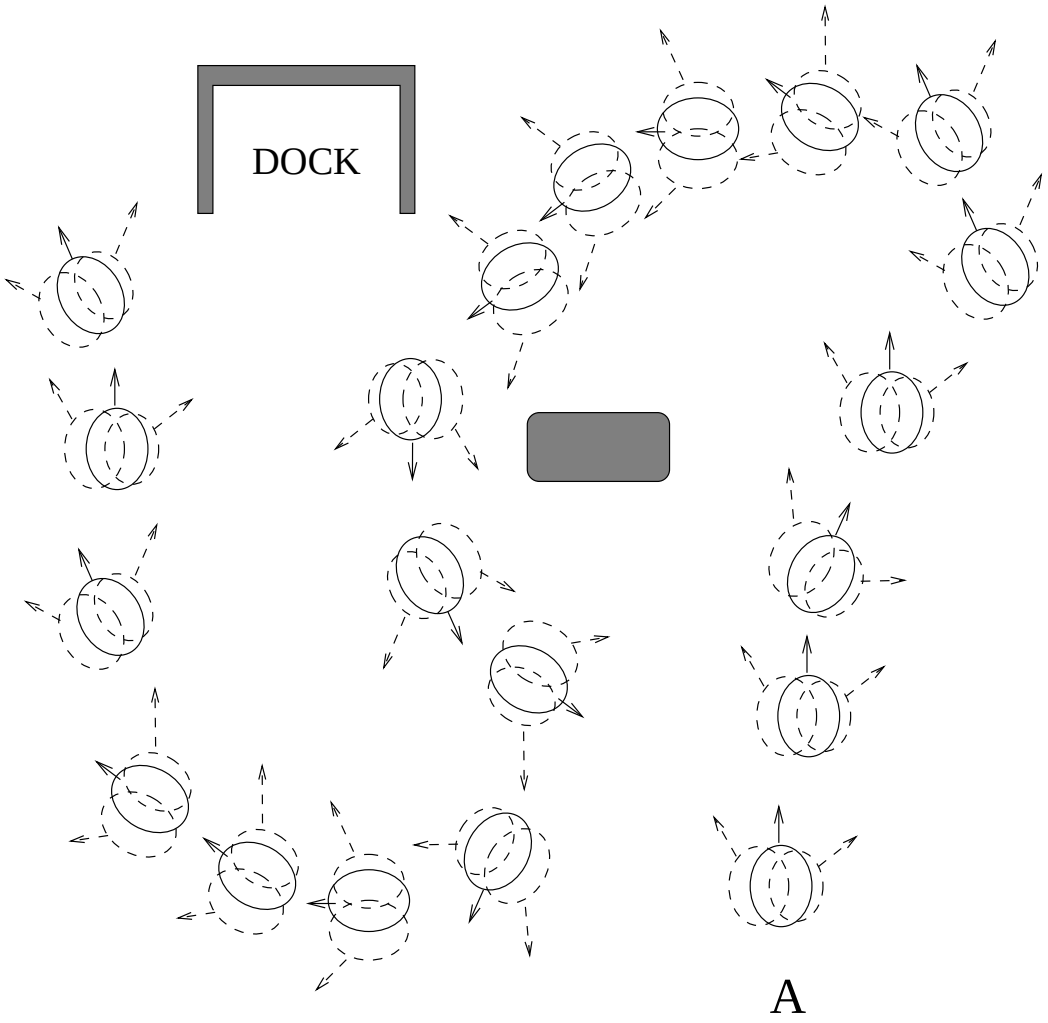


Figure 15: A sample of the path the robot might follow during reinforcement learning. The robot makes moves probabilistically and explores the environment. Shown is an episode of training that results in a failure, the robot never reaches the DOCK. For each move the robot receives a reinforcement of -1 if the state is not the goal state, -100 if it hits an object and +100 if it reaches the goal. RL may consist of a number of such episodes before the robot learns to reach the DOCK.

of moves the robot should make before stopping if it does not reach the goal.

The results obtained from each experiment were averaged to get a final result of the the performance of the robot. Table 3 shows the average results for the five experiments. Dynamic objects like people affected the evaluation, causing the robot to wander a lot and ending up with a collision. Figure 16 shows the percentage of collisions and successes in reaching goal by RuleNavigation, RL and RL with Teacher. The percentage of the robot reaching the goal is low initially and then increases with the training episodes. It finally reaches a maximum of 80% at the end. This percentage is an average taken over the 5 experiments.

5.3 RL With RuleNavigation as a Teacher

RuleNavigation was also used as a teacher. During a run of RuleNavigation, the robot stored state descriptions with the action taken in that state. A number of state descriptions are stored, collectively called a lesson, and used when RL is applied to the robot. The robot refers to the stored states and action when making a move. This results in the reduction of random moves made by the robot. The presence of a teacher affects the learning by speeding it up. Table 4 shows the average results over the 5 experiments for RL with a teacher.

The number of steps taken by the RuleNavigation was generally 10. The robot just turned in the direction of the goal and moved towards it. The stored moves are then replayed as if the robot is learning to move on that path only. The training and evaluation is done in the same way as in the RL without teacher after the replay is over. The robot makes random moves to reach the goal. If it finds the path followed in the replay it is pushed towards the goal.

For RL with the teacher after a training phase an evaluation phase was com-

Table 3: Below are the average results for the evaluations in 5 experiments done for RL without a teacher. The average performance of the robot over the 5 experiments improves with the training and reaches a maximum of 80%.

Evaluation No.	Runs	Percentage of Times Goal Reached	Average Steps before Collision	Average Steps to Reach Goal
1	5	20	14.34	11
2	5	0	9.76	na
3	5	60	11.2	12.67
4	5	0	10.76	na
5	5	20	9.84	13
6	5	0	11.56	na
7	5	0	6.16	na
8	5	40	11.08	12
9	5	20	9.53	11
10	5	40	14.31	11.3
11	5	20	10.84	12
12	5	20	8.1	15
13	5	50	13.82	9.5
14	5	60	16.24	10.08
15	5	45	15.07	9.58
16	5	53.33	15.7	9.67
17	5	66.67	15.6	9.67
18	5	65	14.68	10.25
19	5	76	15	10.3
20	5	80	15	10.3

Table 4: Results for RL with a teacher. The results are again averaged over the 5 experiments. The performance of the robot is certainly better with the teacher, which speeds up the learning and promises a faster convergence to the solution.

Evaluation No.	Runs	Percentage of Times Goal Reached	Average Steps before Collision	Average Steps to Reach Goal
1	5	20	11.53	7
2	5	20	14.16	21
3	5	60	17.20	10
4	5	33.33	13.88	12.167
5	5	60	15.74	13
6	5	40	13.73	12.8
7	5	100	13.2	10
8	5	53.33	13.65	11
9	5	60	15.96	10.167
10	5	60	14	10.33
11	5	72	15	10.2
12	5	80	15.3	10
13	5	80	15.3	10
14	5	80	15.67	10
15	5	80	15.67	10
16	5	80	15.67	10
17	5	80	15.67	10
18	5	80	15.67	10
19	5	80	15.67	10
20	5	80	15.67	10

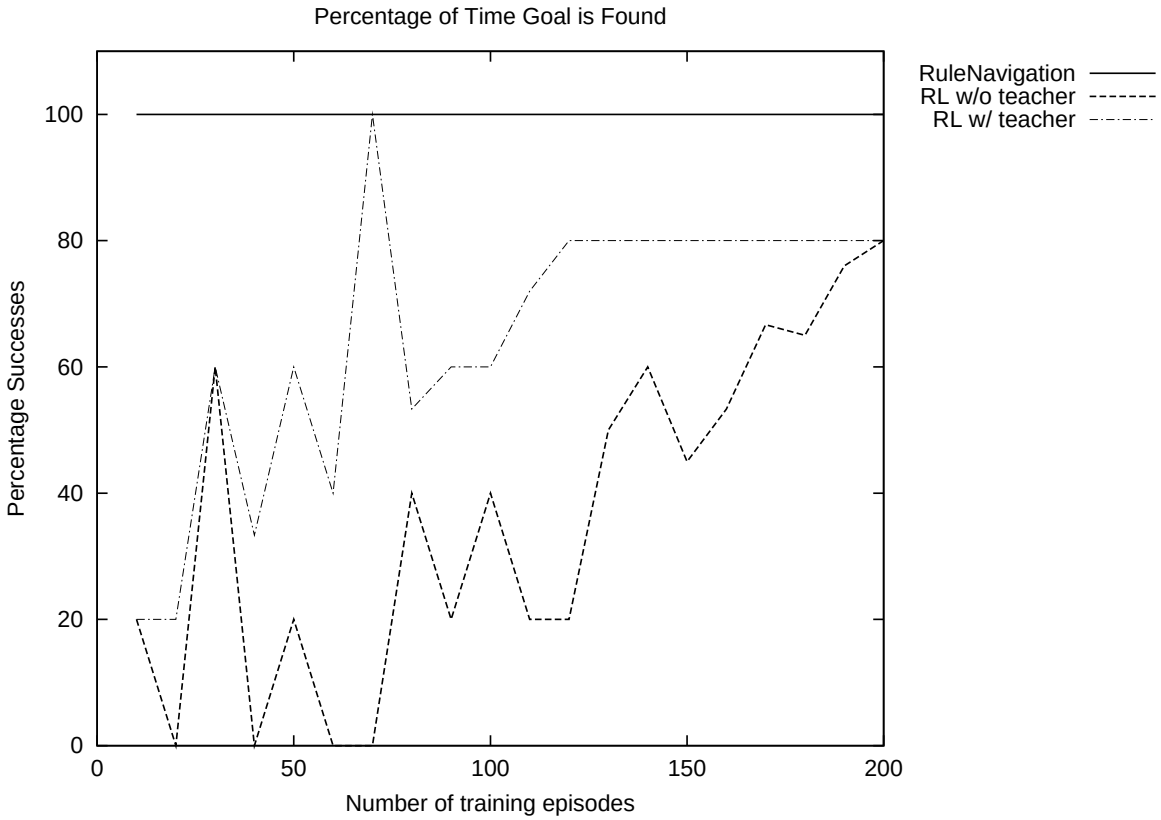


Figure 16: Graph of percentage of times goal is found for the two learning methods and RuleNavigation averaged over the five experiments conducted. The robot reaches the goal in the initial training periods but this is not consistent. As the training continues and the learning becomes more refined and the robot starts reaching the goal more consistently (shown by the straight line towards the end). The graph has been plotted for RL and RL with a teacher. The RL with teacher performs better than the RL method. In both the methods the robot reaches the goal 80% of the times at the end of the training episodes.

pleted. The performance of the robot was as good as or better than RL without teacher through all the evaluation phases. We can see in Table 4 at the RL with a teacher converges to a solution faster than RL. The average time for running the training phase and evaluation was the same as the earlier experiments with RL. Figures 17 and 18 show the number of steps before collisions and the number of steps to reach the goal.

5.4 Evaluation of the Results Obtained

The RL results obtained show that a lot of time is taken to learn a task. Even a simple task such as moving between two points may take more than 20 training phases. The number of runs before the goal is robot actually reached the goal during evaluation was high. The performance was affected dynamic objects like people walking around it. As the robot learns the number of steps before it collides with the walls increases. The number of times it reaches the goal during evaluation also increases. With the help of a teacher the convergence is faster.

The results obtained in RL with RuleNavigation are definitely better than the ones obtained when using RL alone. The reason for this is that in RL the robot makes random moves to reach the goal. It is not guided by any external agent towards the goal and must explore the environment by making all possible moves. By making random moves it learns from the reinforcement or the reward received, whether its move was a good one or a bad one. So, if in a state it made a bad move earlier, we are sure that it will not make that move again. By exploiting actions already taken and exploring the environment by taking new ones the robot moves towards a better policy which would tell the best move to take when in a particular state. The question is how long would it take to learn that policy?

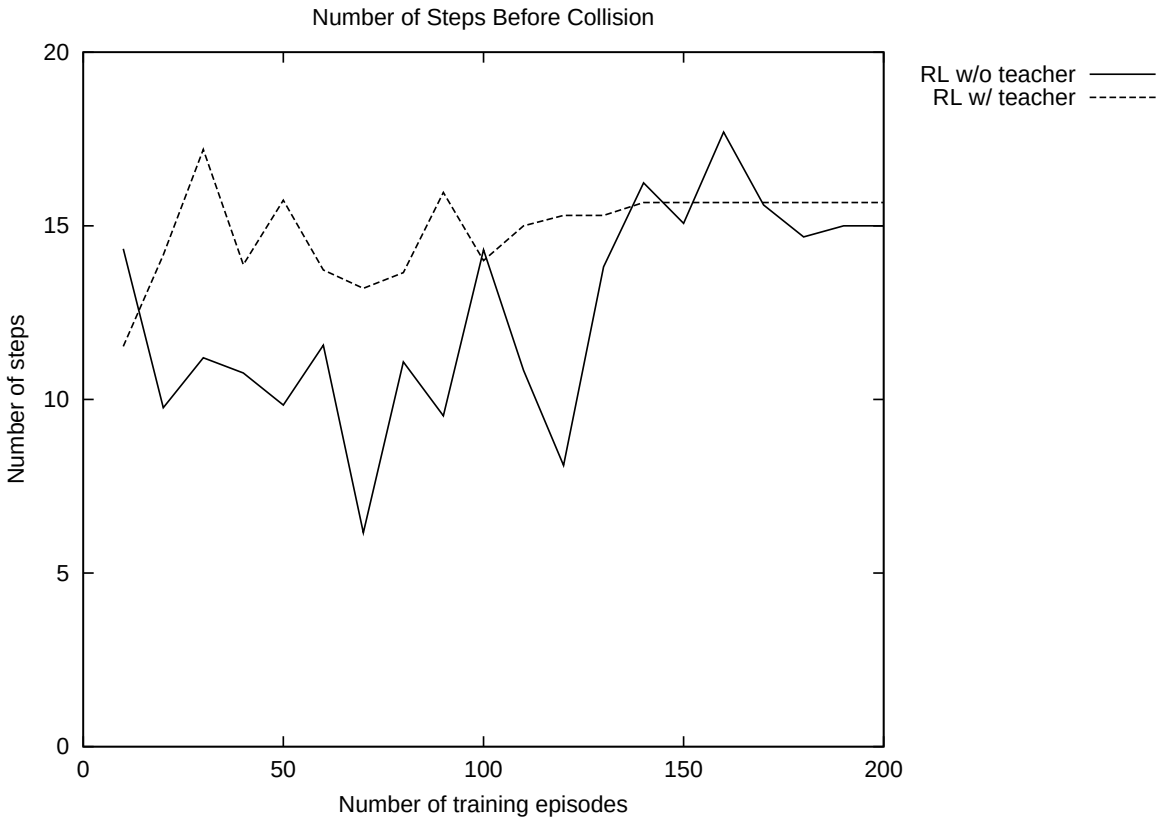


Figure 17: Graph showing number of steps before collision after training. Collisions occurred later after increased training. The number of steps before collision increase as the robot learns how to avoid obstacles and make the right moves to avoid them. The number of steps taken before collision are expected to increase as the learning continues. It may drop to zero if the robot reaches the goal all the times in the testing. The increase in the number of steps before collision is a sign that the robot is learning to avoid obstacles while moving around in the environment.

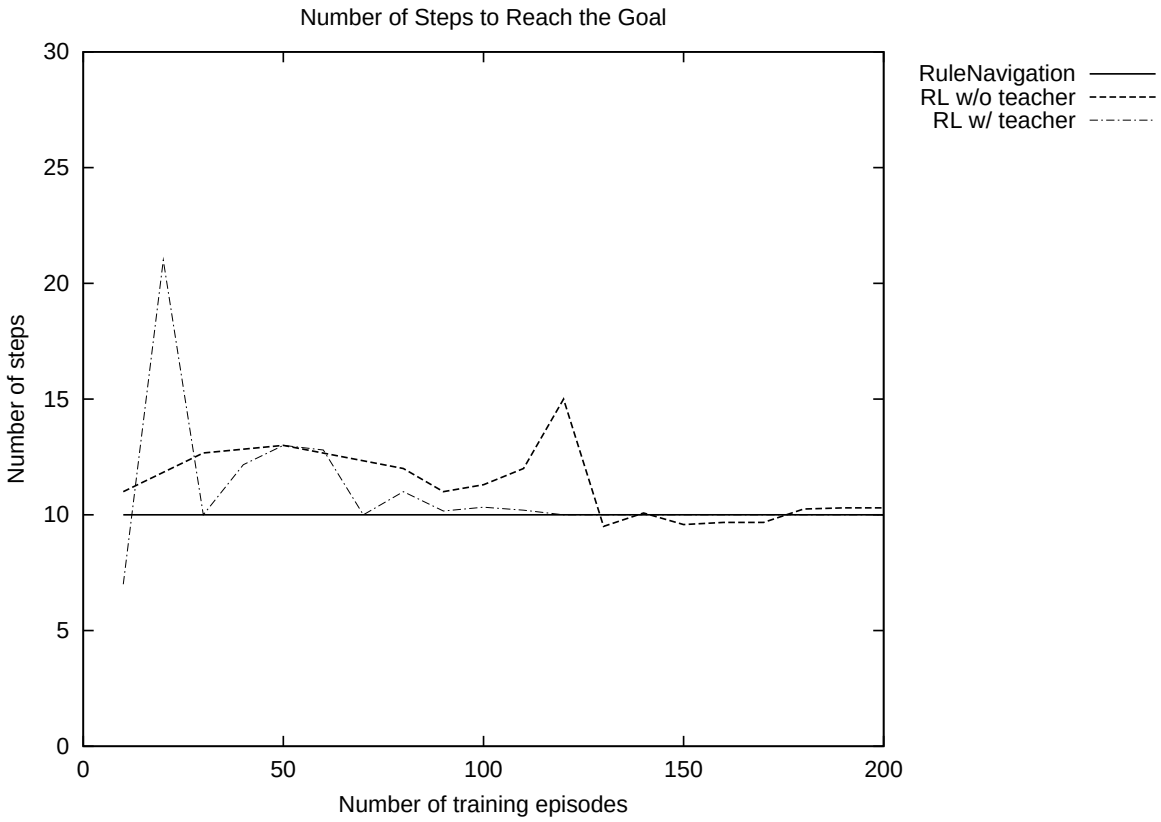


Figure 18: Graph showing number of steps taken to reach the goal after training. Initially the robot may not reach the goal and if it does it takes more than the required number of steps. With training the robot learns the best path to the goal. Once the task is learned the number of steps to reach the goal reduce and finally become constant.

In RL with teacher the agent has a guide to take it towards the goal. If it encounters a state common to the recorded lesson, it will be directed towards the goal. This ensures that the learning would be faster with a teacher than without it.

6 Conclusions

Reinforcement Learning (RL) has been applied to robot navigation problems. The aim is to find the best policy which would map states to actions. In a real world environment, applying RL to an agent can be time consuming. The convergence of a policy to the best one is slow and it may take thousands of episodes before the agent learns a task perfectly. A method of speeding the convergence of the RL algorithm is replay. In replay previous experiences are stored and the agent makes use of the past experience to make better moves during learning. The agent has the experience of the teacher which guides it towards the goal and helps in learning the task faster. The question is whether the learning time can be reduce and whether replay is effective in doing so.

6.1 Thesis

In my thesis I have investigated a form of RL called Q-learning. Q-learning is effective in teaching a task but as seen requires a lot of training time. As seen in the experiments the robot does not reach the goal until 8 training phases of 10 runs each have passed. Even after that there is no consistency in reaching the goal. This indicates the need for more training phases. Thus Q-learning will converge to a solution but is slow. The results for RL with a teacher show that learning can be hastened by using replay. With a teacher Q-learning converges faster to a solution. The robot reached the goal after 30 runs.

6.2 Results

The results obtained show that automatic methods such as RL take a long time to converge. The number of episodes done were 200 for RL without the teacher. The number of times the robot reached the goal was small. The number of times the robot collided was high. The low percentage of goal reaching runs indicates more training runs are needed. The time required for training and evaluation was high. This means that if a more complex task was to be performed the time taken to learn it would increase.

If the teacher is used a quicker solution is possible. The teacher reduces the difficulty of learning, as seen earlier where the robot began reaching the goal after 3 training phases as compared to 8 training phases in case of using RL only.

6.3 Future Work

Robot navigation is a basic task in learning. We have seen that RL applied to such tasks takes a lot of time. One future possibility is applying RL to more complex tasks such as navigating from one room to the other. Complex tasks require a better description of the environment. For instance internal maps [Thrun and Buecken, 1996, Wolfram et al., 1999] can be created or a camera can be used to identify objects. A task can be broken into subtasks and each subtask to be learned to learn the entire task. We can assume that the learning of a subsequent subtask is begun only when the agent has learned the present one. In replay instead of storing all the states we can store some of them. Also a neural network representation can be used for Q-learning instead of a table.

References

[Chambers and Michie, 1968] Chambers, R. and Michie, D. (1968). Boxes: An experiment in adaptive control. *Machine Intelligence*, 2.

[Chapman and Kaelbling, 1991] Chapman, D. and Kaelbling, L. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. *In Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 726–731.

[Choset and Burdick, 1995] Choset, H. and Burdick, J. (1995). Sensor based planning, part ii: Incremental construction of the generalized voronoi graph. In *Proceedings of the 1995 IEEE International Conference on Robotics and Automation*, pages 1643 – 1648.

[Giacomo et al., 1996] Giacomo, D., Iocchi, L., Nardi, D., and Rosati, R. (1996). Moving a robot: The KRR approach at work. In *Principles of Knowledge Representation and Reasoning*, pages 198–209.

[Giacomo et al., 1997] Giacomo, D., Iocchi, L., Nardi, D., and Rosati, R. (1997). Planning with sensing for a mobile robot. In *Proceedings of the European Conference on Planning*, volume 1348, pages 156–168.

[Gonzalez et al., 1999] Gonzalez, H., Mao, E., Latombe, J., Murali, T., and Efrat, A. (1999). Planning robot motion strategies for efficient model construction. In *Proceedings of the 9th International Symposium of Robotics Research.*

[Group, 1997] Group, M. C. (1997). Intoduction to EBX. Technical report, Motorola.

[Lin, 1993] Lin, L. (1993). Scaling up reinforcement learning for robot control. *ICML (International Conference on Machine Learning)*, pages 182–189.

[Maclin and Shavlik, I 94] Maclin, R. and Shavlik, J. (AAAI-94). Incorporating advice into agents that learn from reinforcements. In *Twelfth National Conference on Artificial Intelligence (American Association of Artificial Intelligence).*

[Mahadevan and Connell, 1991] Mahadevan, S. and Connell, J. (1991). Automatic programming of behavior-based robots using reinforcement learning. *Aritificial Intelligence*, 55(2):311–365.

[Michi and Vinoski, 1999] Michi, H. and Vinoski, S. (1999). *Advanced CORBA Programming with C++.* Addison-Wesley Pub Co.

[Millãn and Torras, 1992] Millãn, J. and Torras, C. (1992). A reinforcement connectionist approach to robot path finding in non-maze-like environments. *Machine Learning*, 8(3/4):363–395.

[Mitchell, 1997] Mitchell, T. (1997). *Machine Learning.* Mc-Graw Hill.

[RWII, 1998a] RWII (1998a). Magellan Pro compact mobile robot user's guide. Technical Report 3625, REAL WORLD INTERFACE.

[RWII, 1998b] RWII (1998b). Mobility robot integration software user's guide. Technical Report 3625, REAL WORLD INTERFACE.

[Saffiotti, 1997] Saffiotti, A. (1997). The uses of fuzzy logic in autonomous robot navigation: a catalogue raisonne. Paper TR/IRIDIA/97-6, Université Libre de Bruxelles, Brussels Belgium.

[Saffiotti et al., 1999] Saffiotti, A., Ruspini, H., and K.Konolige (1999). Using fuzzy logic for mobile robot control. In *International Handbook of Fuzzy Sets and Possibility Theory*. Kluwer Academic Publisher.

[Simmons, 1996] Simmons, R. (1996). The curvature-velocity method for local obstacle avoidance. In *International Conference on Robotics and Automation*.

[Sutton and Barto, 1998] Sutton, R. and Barto, A. (March 1998). *Reinforcement Learning: An introduction (Adaptive Computation and Machine Learning)*. MIT Press.

[Thrun and Buecken, 1996] Thrun, S. and Buecken, A. (1996). Learning maps for indoor mobile robot navigation. Paper 121, Computer Science Department, Carnegie Mellon University.

[Watkins and Dayan, 1992] Watkins, C. and Dayan, P. (1992). Q-learning. *Machine Learning, 8*, pages 279 – 292.

[Wolfram et al., 1999] Wolfram, B., Fox, D., Jans, H., Matenar, C., and Thrun, S. (1999). Sonar-based mapping with mobile robots using em. In *Proceedings of the 16th International Conference on Machine Learning*.