An Integrated Learning, Planning and Reacting Algorithm Applied to a Real Mobile Robot

ALEX WEISER
Lehrstuhl für Prozeßrechnertechnik
Technische Universität München
80333 München, Germany
PEDRO LIMA
Instituto de Sistemas e Robótica
Instituto Superior Técnico - Torre Norte
Av. Rovisco Pais, 1
1096 Lisboa Codex, Portugal

Abstract

Sutton's Dyna algorithm for integrated learning, planning and reacting is applied to a real mobile platform (Robosoft's Robuter). The mobile robot uses sonar to scan for obstacles and odometry for self-localization. Practical problems associated with the implementation of the algorithm on a real setup and results from real experiments are presented and discussed.

Keywords: Reinforcement Learning, Planning, Reacting, Mobile Robots, Dyna algorithm.

1 Introduction

Many researchers have applied learning to Robotics. A robot can learn from the data provided by its external sensors (e.g. cameras, ultrasound transducers, proximity detectors) or internal alarms (e.g. battery failure, timeout while running a process). *Reinforcement learning*, where limited information is available about the algorithm instantaneous performance, typically in the form of success or failure signals, is particularly interesting for Robotics as it involves the exchange of small bandwidth information (failure and success signals only) between robotic subsystems. In the last few years, Barto, Sutton and their associates have explored reinforcement learning algorithms [1]. Recently, the adaptive behavior community has applied extended versions of Sutton's *TD-learning* [4] and Watkins' *Q-learning* [6] algorithms to Robotics [2, 3].

A frequent limitation of reinforcement learning applications to Robotics is the problem large state space. Lin [2] has attempted to tackle this by providing some initial knowledge to the robot and by endowing it with generalization capabilities using neural nets. Sutton [5] describes an algorithm (Dyna) that learns from experiences with the real world intermixed with virtual experiences done on an internal world model to speed up the learning process. Both authors use Watkins' *Q-learning* algorithm, which includes a performance function [6]. A drawback of these approaches is the fact that they rely on a direct and error-free evaluation of task success. This is not always realistic in real implementations, since checking if the goal was reached or navigating through a world full of obstacles are usually difficult and error prone tasks.

This paper describes an implementation of Sutton's Dyna algorithm [5] on a real mobile robot, the Robuter platform built by the French company Robosoft. The mobile robot uses sonar to scan for obstacles and odometry for self-localization. Practical problems associated with the implementation of the algorithm on a real setup and results from real experiments are presented and discussed. The task consists of navigating a mobile robot through a maze, from an initial to a goal position. The maze is initially unknown to the mobile robot. We also discuss possible extensions which may be helpful to study the application of this type of reinforcement learning algorithms to situations more realistic than the simulations usually described in the literature.

In Section 2 there is an overview of the algorithm, followed by two Sections containing a description of the mobile robot used, and of the implementation of the algorithm in a real setup. Finally, results and conclusions are presented in Sections 5 and 6, respectively.

2 Dyna Algorithm

2.1 The Model

The Dyna algorithm is based on three modules. Two of them are the *world* (or *world model*) and the *policy*. This part of the algorithm works like a reactive system. The policy associates to each world state an action that moves the robot to a new world state.

In addition, an *evaluation* is calculated for each state according to successive reinforcement signals received from the world after each move. The learning scheme updates the policy and the evaluation functions. By updating its policy, the robot is effectively planning a set of reaction rules to be used in the next step.

In the application described here, the world is represented by a grid of square cells (states); therefore, the world model is a cell matrix. The policy contains for each cell a probability for each action that moves the robot to one of the four neighbor cells. The evaluation is also a matrix containing an entry for each cell.

2.2 Real and Hypothetical Experiences

A *step* is a motion from one cell to a neighboring cell. Hypothetical and real steps are performed. In a real step the robot moves from one position to another and provides new information to the world model. A hypothetical step, however, is a simulated step in the world model. Both real and hypothetical steps use the same algorithm, updating the policy and evaluation through a learning process. A *trial* consists of a sequence of real steps from the starting point to the goal. A fixed number of hypothetical steps is performed between real steps.

2.3 The Algorithm

An outline of the Dyna algorithm follows[5]:

1. Decide if it is a real step or a hypothetical one.
2. Pick a state x. If this is a real step, use the current state. If this is a hypothetical experience, choose a random state that is already in the world model.
3. Form prior evaluation of state x according to the result of the evaluation function: $e = Eval(x)$.
4. Choose an action a to be taken from state x by consulting the policy.
5. Execute the action a if there is no obstacle in this direction in the world and world model. If the action is executable, obtain next state y and reward r_y from world or world model.

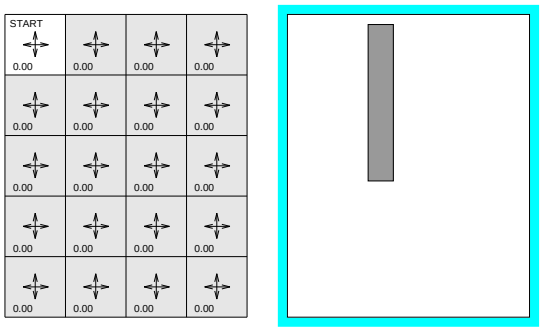

Figure 1: Initial values

6. Form posterior evaluation e' of the old state x using the evaluation of the new state y now reached: $e' = r_y + \gamma Eval(y)$.
7. If this is a real step, update x and r_y in the world model.
8. Update the evaluation function so that the evaluation of the old state x is closer to the posterior evaluation e' rather than to the prior evaluation e.
9. Update the policy. Strengthen or weaken the tendency to perform the action a from state x again according to the difference between the posterior and prior evaluation: $e' - e$.
10. Go to step 1.

The policy matrix has an entry w_{xa} for every pair of state x and action a, i.e., four entries for each state in the world model, corresponding to actions up, down, left, right. Actions are chosen randomly based on a Boltzmann probability distribution, obtained by turning the four policy values for a given state in probabilities using . The probability for each action a from state x is then represented by $P\{a \mid x\}$ according to Equation(1):

$$P\{a \mid x\} = \frac{e^{w_{xa}}}{\sum_j^{\text{actions}} e^{w_{xj}}} \tag{1}$$

The evaluation table is updated according to the simplest version of the temporal difference learning method:

$$Eval(x) = Eval(x) + \beta(e' - e). \tag{2}$$

Figure 1 displays all the information available to the algorithm. It shows a sample model with the dimensions of 4x5 cells. The arrow lengths indicate the probability to execute an action. The number in each cell represents the evaluation. After initialization the action probability is uniformly distributed and the

evaluations are zero. A white cell is a KNOWN (OCCUPIED — when crossed — or EMPTY) cell, whereas the dark cells are UNKNOWN. When the robot moves in an environment like the one represented in the right side of Figure 1, during the second trial the information may look like what Figure 2 shows, with OCCUPIED cells and still UNKNOWN cells. The evaluation increases towards the goal and the policy brings the system to the goal from any empty cell. The policy becomes weaker the higher the distance from goal because the evaluation differences are lower. The first trial proceeds in a completely random fashion. After the goal is reached for the first time, a reward is received, making the goal cell attractive to neighboring cells. This "attraction" to the goal cell is backpropagated to the start cell by a dynamic programming algorithm, embedded in the TD-learning method.

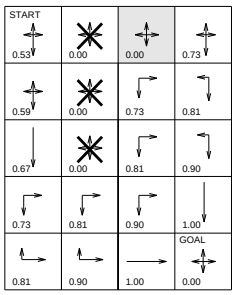

Figure 2: Example with adjusted values

3 Using a Real Mobile Robot

3.1 The Robuter Platform

Experiments with the real robot requires attention to some additional points that are ignorable during simulation. The real robot calculates its position through odometry, detects obstacles by sonar scanning, and needs to execute an evasive action when it finds an obstacle while moving from one position to another. The robot has an acceleration and deceleration phase in the movements and it cannot stop instantly. There are errors in the sensor readings that must be taken into account. The robot must return to its start position at the beginning of a new trial. Finally, the localization of the goal may be a difficult task requiring complex sensing and reasoning.

The robot, shown in Figure 3, consists of a mobile rectangular platform with the dimensions of 1000x700mm. It has two independent propulsive wheels at the back and two free rotating and free turning wheels at the front. Both back wheels are

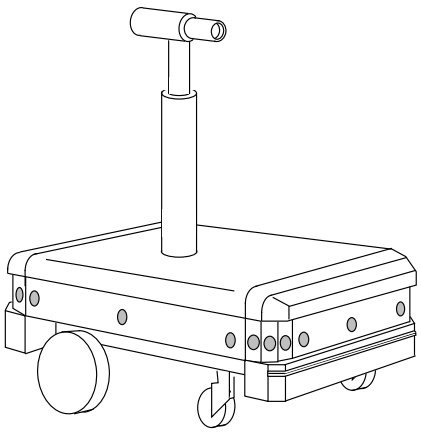

Figure 3: Robuter Platform

equipped with encoders for positioning. The rotation axis is located over the center of the driving axis. There are 24 ultrasonic transducers positioned around the vehicle for obstacle detection, and also an infrared camera.

3.2 Albatros Operating System

The Robuter's operating system is *Albatros*, a real-time operating system specifically designed to control, with minimal hardware and maximum efficiency, any kind of multi-axis and multi-sensor device. It has a multi-tasking organization with predefined priorities and fixed scheduling for tasks.

The application program can share the same processor with the Operating System allowing the use of its resources through system calls. It is possible to link the *Albatros* kernel with the user's specific programs.

4 Experimental Setup

The size of a cell is shown Figure 4. Although the length of the platform is only one meter, the minimum size of a cell must be equal to 4 m^2. When the robot is positioned in a cell its rotation axis should be in the center of the square to give the robot space enough to turn around without crossing the border of the cell. In addition, there is a safety margin of 20 cm.

To detect the obstacles, the robot uses the three sensors located on its sides, respectively. The values returned from a sensor correspond to the time elapsed between emission and reception of an ultrasonic wave. The distance can be approximately computed through the returned value. The returned value however changes depending on the type of the objects, external temperature, etc.

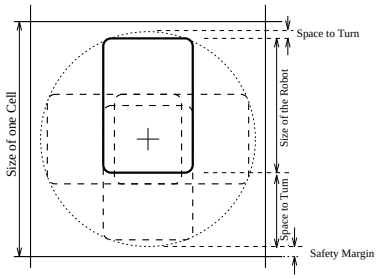

Figure 4: The Size of a Cell

The calibration of the sonar was made by positioning the objects at a desired distance and making a sensor data read. This value was then used as a threshold. All objects situated closer to the robot than the threshold distance are identified as obstacles and therefore make the robot stop.

To keep the odometry failures low, sonar scans were made prior to any motion. Before the robot starts a movement it scans the direction it is going to move along. In this scan, the threshold reaches the middle of the neighbor cell in order to check for an obstacle.

Once the robot begins to move, the threshold decreases proportionally to the distance traversed in order not to cross the border of the target cell, thereby avoiding the detection of an obstacle located in the cell after the one the robot is moving into.

The position of the robot is only determined by odometry: therefore, the decrease of the sonar threshold is computed from the encoder readings. Prior to every movement the encoders are reset. If at the end of a move there is an offset between the desired position and the actual robot position, this offset is subtracted from the desired distance in the next motion step.

Because the rotation axis of the robot is not located over the center of the platform the threshold used cannot be the same for all directions relative to the robot. Thus, a margin is added when the direction to scan is not the front side.

Figure 5 shows a flow chart of the implemented algorithm. In the flow chart, the `MotionProcess` is a routine running as an independent and interruptible process. It starts every 10 clock cycles, handling all the communication with *Albatros* concerning the translations and rotations, the emergency stop, controls the sonar scanning and continuously updates the status of the movement.

The `RelPosError` refers to the error of the position of the robot relative to the desired end position.

The `MotionProcess` runs parallel to the main process of the application program and communicates with it through a common area. This has the advan-

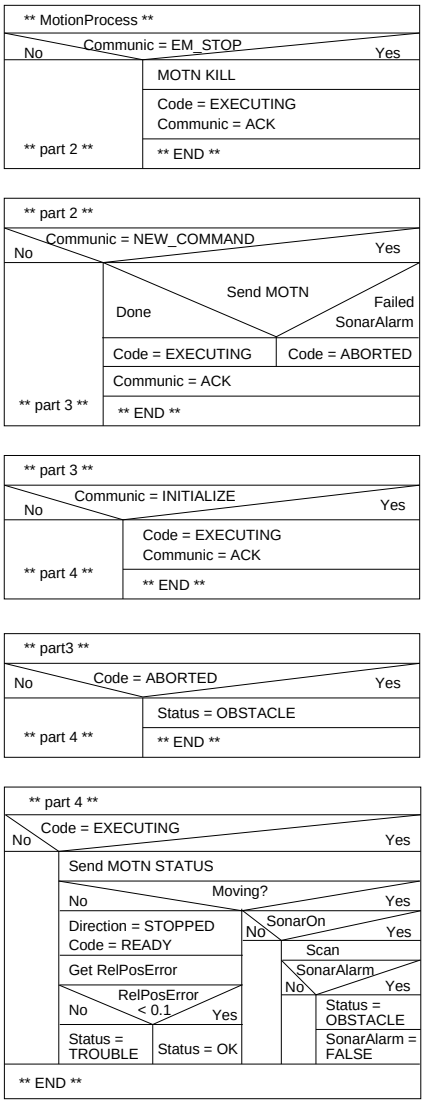

Figure 5: Flow Chart of MotionProcess

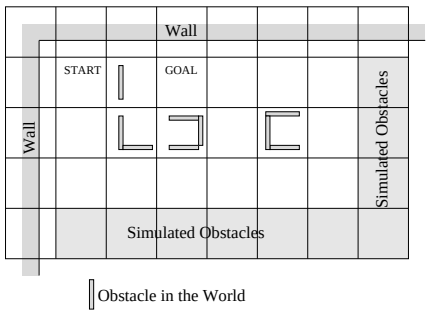

Figure 6: Obstacles in the World

tage that the motion can be stopped at any time by another process or any external sensor that uses the same common area. The main process could execute another tasks while the robot is moving with the evasive action being controlled by the motion process.

5 Results

The more hypothetical steps are performed, the faster is the learning process. The learning algorithm always converges without getting caught in local minima. The random number generator has a large influence in the duration of the experiment since it is used to choose the actions and to determine the states for the hypothetical steps.

An environment with obstacles positioned in the world, as shown in Figure 6, is used for an experiment. The walls of the room are used as obstacles. Wood fiberboard of the size of 120x50cm is positioned vertically representing additional obstacles. Some obstacles are still simulated to avoid the robot running into obstacles that can not be detected by the sonars, e.g stairs, and to prevent the system from leaving the boundaries of the world model.

Below are the results of the experiments for three trials using 50 hypothetical steps. The numbers indicate the number of visits of the robot to each cell, and the sign # represents an identified obstacle.

```
TRIAL 1
     #
 #   3       1   1
 #  11   #   #   1
 #   5   2   2   2
                 #
 28 steps 21 calls 16 motions

TRIAL 2
     #
 #   8   #   1   1
 #   4   #   #   1
 #   1   1   1   1
                 #
 19 steps 11 calls 12 motions

TRIAL 3
     #
 #       #   1   1
 #   1   #   #   1
 #   1   1   1   1
                 #
 8 steps 8 calls 8 motions
```

In this run, the shortest path was found in 3 trials, requiring 8 steps to reach the goal.

The implementation showed the difficulty of implementing the algorithm in a real mobile robot. It is necessary to control the robot motion keeping a record of its location through odometry, to scan for obstacles, to perform evasive actions, and to deal with inaccurate sensor data and accumulated errors. The results of the algorithm, however, are exactly the same as the ones obtained in the simulation, except that it was necessary to reset the robot position once every 3–4 trials, due to the accumulation of odometry errors.

In a real setup, the robot needs not move on every real step. Some steps can be blocked via hierarchically higher software levels, when an obstacle is detected, consulting the world model or using sonar scans before starting to move the robot in the direction of the obstacle, as explained above.

6 Preliminary Conclusions and Future Extensions

In this paper we attempted to verify a concept and its functionality by using an internal world model built from real sensors to speed up the incremental learning by a mobile robot, of a set of reaction rules which allow it to find the shortest way out of a maze.

In future work, a set of modifications and additional functions can be implemented to improve the performance of the robot:

- The simple search method used here combined with the simple reward make the first trial very time consuming, with low information gain. The system is not learning and planning and the robot runs aimlessly until it happens to find the goal. To speed up the first trial one could make use of other approaches, such as lateral scanning instead of scanning only in the direction of movement, storing the detected obstacles in the world model.

- The policy might be changed when the robot bumps into an obstacle, using a different learning process without modifying the evaluation of a cell. Currently, reinforcement consists of a reward only when the goal is reached, and no penalties (e.g., when an obstacle is detected ahead of the robot).

- An attempt can be made by using bigger virtual cells in the beginning of an experiment just to find the goal more quickly and later divide every cell of the world into smaller ones. That, however, may contradict the idea that , once an obstacle has been located in a cell, the entire cell is assumed to be occupied until the end of the algorithm.

  The use of beacons and an infrared camera can help the robot to improve its self-localization. By turning the camera while staying at a fixed position, the robot determines the angles ϑ_1 and ϑ_2

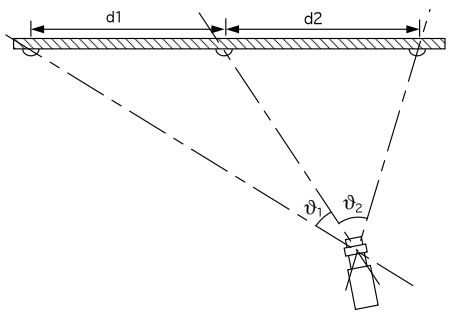

Figure 7: Localization using Infrared Camera

shown in Figure 7, and the robot position can be determined by triangulation.

Large odometry failures can shift or twist the world model in such a way that it simply doesn't model the real world anymore. The robot then sees an obstacle in one cell, which in the beginning was located in a neighboring cell. The infrared camera and beacons can be used to correct the position of the robot and therefore the odometry failures. Goal self-localization would then be a particular example of this application.

- The use of parallel processing by performing the hypothetical steps simultaneously to a real step is not very interesting for this experiment specifically due to the robot used. However, when using a smaller and quicker robot such an approach would be interesting. Actually, a fleet of small robots running in parallel through the maze, while updating a centralized world model, would be another interesting extension of this work, with the objectives of speeding up the algorithm and testing strategies of cooperation.

Acknowledgments

This work was partially supported by the grant PRAXIS/3/3.1/TPR/23/95. The first author would like to thank Dipl.-Ing. Alexa Hauck from Lehrstuhl für Hochfrequenztechnik for her constant support throughout this work.

References

[1] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, SMC–13(5):835–846, 1983.

[2] Long-Ji Lin. Scaling up reinforcement learning for robot control. In *Proceedings of the Tenth International Conference on Machine Learning*, 1994.

[3] S. P. Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8:323–339, 1992.

[4] R. S. Sutton. Learning to predict by the methods of Temporal Differences. *Machine Learning*, 3:9–44, 1988.

[5] R. S. Sutton. First results with Dyna, an integrated architecture for learning, planning and reacting. In *Neural Networks for Control*. The MIT Press, 1990.

[6] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.