11
Exploration of an unknown environment by a mobile robot

Rui Araújo, and A. T. de Almeida
Institute for Systems and Robotics (ISR), and
Electrical Engineering Department, University of Coimbra
Pólo II, Pinhal de Marrocos, 3030 Coimbra, Portugal
Telephone: +351 (39) 7006276/00. Fax: +351 (39) 35672.
email: rui@isr.uc.pt

Abstract

Mobile robots can make important contributions to the integration of industrial operations. The improvement of mobile robot autonomy will lead to many benefits which include a decrease in programming effort, simplifications on shop floor design, and the lowering of costs and time to market. Additionally this will free human resources which may be applied to the areas of product development. All those factors will ultimately contribute for enabling more agile and sustainable industrial production. In this article we demonstrate the validity of the parti-game self-learning approach for navigating a mobile robot, by finding a path to a goal region of an unknown environment. Initially, the robot has no map of the world, and has only the abilities of sensor-based obstacle detection and straight-line movement. Simulation results concerning the application of the learning approach to a mobile robot are presented.

Keywords

Learning control, path finding, mobile robot

1 INTRODUCTION

An important attribute for autonomous robots, is the ability to navigate their environments, which are usually unknown the first time robots face them. In this article we treat the problem of controlling an autonomous mobile robot, so that it reaches a goal location in an unknown 2D environment. This is called the *robot path finding problem* where a path that avoids collisions with obstacles must be generated between an initial and a goal robot configurations.

Many control methods are based on mathematically modelling the system/plant to be controlled. In the case of mobile robotics for example, it is usual to assume world knowledge, generally appearing in the form of a *global map* of the world over which *path planning* algorithms operate. However it is difficult to provide the robot with a global map model of its world. The difficulties may arise from various reasons. The *map building* operation is itself a difficult separate problem. It may become even more complicated if

L. M. Camarinha-Matos (ed.), *Re-engineering for Sustainable Industrial Production*

the robot environment changes. Also, if the robot control system requires the introduction of the global map, this may become a tedious and time consuming programming task.

On the other hand, *reactive systems* (e.g. (Brooks 1986)) don't require the existence of a global map. Those systems are organised as a layered set of task-achieving modules. Each module implements one specific control strategy or *behaviour* like "avoid hitting anything" or "keep following the wall". Each behaviour implements a close mapping between sensory information and actuation to the system. However, basic reactive systems suffer from two shortcomings. First, they are difficult to program. Second and most important, pure reactive controllers may generate inneficient trajectories since they choose the next action as a function of the current sensor readings, and the robot's perception range is limited. To address this second problem, some control architectures combine *a priori* knowledge and planning, with reaction (e.g. (Arkin 1990)).

Robot programming and control architectures must be equipped to face unstructured environments, which may be partially or totally unknown at programming time. An interesting possibility, is *learning* to act over the world by self-improvement of controller performance. This may be viewed as a concept of *self programming* in which control of a complex system in principle does not need extensive analysis and modelling by human experts. Instead the system makes use of sensory information, and learns to interact with the world by constructing it's own abilities.

In this paper we demonstrate the effectiveness of the Parti-game self-learning approach (Moore & Atkeson 1995) to the specific case of learning a mobile robot path from its initial position, to a known goal region in the world. Also the algorithm does not have any initial internal representation, map, or model, of the world. In particular the system has no initial information regarding the location or the shape of obstacles. We demonstrate that the mobile robot can learn to navigate to the goal having only the predefined abilities of doing straight-line motion, and obstacle detection (not avoidance) using its own distance sensors.

The organisation of the paper is as follows. Section 2 presents the learning controller architecture. In section 3 we present the experimental environment around which we performed some experiments concerning the application of the exploration algorithm. Section 4 presents results of simulations using the controller for learning to navigate the robot to a goal region on an unknown world. Finally in section 5 we make some concluding remarks.

2 LEARNING CONTROLLER ARCHITECTURE

The problem we wish to solve in this work may be stated as follows: The mobile robot is initially on some position on the environment, or world, and then it must learn a path to a known goal region in the world. Also the algorithm does not have any initial internal representation, map, or model, of the world. In applying the algorithm, we assume that the initial abilities of the mobile robot are only two. First, it is able to perform *obstacle detection* operations, i.e. to detect obstacles that may obstruct its normal path. Second, the mobile robot is able to perform *straight line movements* between its current position and some other specified position in the world. Two comments to this second ability are in order. First, this ability implies the knowledge of the robot current position, and second the movements may fail because of the presence of an obstacle that is detected.

The mobile robot controller architecture used in this work, is based on the application of the parti-game learning controller algorithm (Moore & Atkeson 1995) to the specific

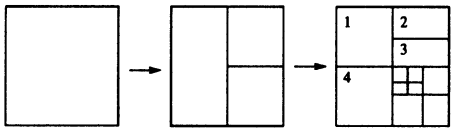

Figure 1 Subdividing the state-space.

case of learning a mobile robot path. In this paper we demonstrate the effectiveness of this approach to the above stated problem.

The parti-game algorithm has application to difficult learning control problems in which: (1) We have continuous and multidimensional state and action spaces; (2) "Greedy" techniques (e.g. trying to act in opposition to an error) and hill-climbing techniques (e.g. optimisation-based training of a neural networks) can become stuck, never reaching the goal; (3) random exploration can be intractably time-consuming; (4) We have unknown, and possibly discontinuous, system dynamics and control laws.

Additionally the algorithm has some restrictions that however, do not prevent its application to our problem: (1) The dynamics are deterministic. (2) The task is specified by a known goal region on a multidimensional state-space. (3) A feasible solution is found, not necessarily an optimal path according to a particular criterion. (4) A local greedy controller is available, which we can ask to move greedily towards a desired state. However, there is no guarantee that a request to the greedy controller will succeed. On our case for example, it is possible for the greedy controller (the "straight-line mover") to become stuck because of an obstacle found by the robot.

2.1 The Algorithm

The Parti-game algorithm is based on partitioning the state-space. It begins with a large partition. Then it increases the resolution by subdividing the state-space (see figure 1) where the learner predicts that a higher resolution is needed. As usual a *partitioning* of the state-space is a finite set of disjoint regions, the union of which covers the entire state-space. Those regions will be called cells, and will be labelled with integers $1, 2, \ldots, N$. In this paper we will assume that the cells are all axis aligned hyperrectangles (rectangles in our 2D case). Although this assumption is not strictly necessary, it simplifies the computational implementation of the algorithm. A *real-valued state*, s, is a vector of real numbers in a multidimensional space (2D space in our case). Real-valued states and cells are distinct entities. For example the right partitioning on figure 1 is composed of eleven cells, which are labelled with numbers $1 \ldots 11$. Each real-valued state is in one cell and each cell contains a continuous set of real-valued states. Let us define **NEIGHS**(i) as the set neighbours, or cells which are adjacent to i. In figure 1 **NEIGHS**$(1) = \{2, 3, 4\}$. When we are at a cell i, applying an *action* consists on actuating the local greedy controller "aiming at cell j". A cell i has an associated set of possible actions that is defined as **NEIGHS**(i). Each action can thus be labelled by a neighbouring cell.

The algorithm uses an environmental model, which can be any model (for example, dynamic or geometric) that we can use to tell us for any real-valued state, control action, and time interval, what will be the subsequent real-valued state. In our case the "model" is implemented by the mobile robot (which can be real or simulated), and takes the current position, and position command, to generate the next robot position.

Let us define the **NEXT-PARTITION**(s, j) function that tells us in which cell we end up, if we start at a given real-valued state, s, and using the local greedy controller, keep

moving toward the centre of a given cell, j, until either we exit the initial cell or get stuck. Let i be the cell containing the real-valued state s. If we apply the local greedy controller "aim at cell j" until either cell i is exited or we become permanently stuck in i, then

$$\textbf{NEXT-PARTITION}(s, j) = \begin{cases} i \text{ if we became stuck} \\ \text{the cell containing the exit state otherwise} \end{cases}$$

In our work, the test for sticking performs an obstacle detection with the distance sensors of the mobile robot (see section 3). In other systems, sticking could be tested by seeing if the system has not exited the cell after a predefined time interval.

Let **CENTER**(i) be the real valued state at the centre of the hyperrectangle corresponding to cell i. Define **NEXT**(i, k) as the cell where we arrive if, starting at **CENTER**(i), we apply the local greedy controller "aiming at cell k":

$$\textbf{NEXT}(i, k) = \textbf{NEXT-PARTITION}(\textbf{CENTER}(i), k)$$

In general **NEXT**$(i, k) \neq k$ because the local greedy controller is not guaranteed to succeed.

Define the *cell-length* of a, possibly not continuous, path on the state space, as the number of cell transitions that take place as we go through the path. When the system is on the real-valued state s of a cell i, one of the key decisions that the algorithm has to take, is to choose the cell at which the system should aim using the local greedy controller. The method for making this decision is called a *policy*. As a first attempt to solve the problem we could choose always aiming at a next cell such that the shortest path (in terms of cell transitions) to the goal cell is traversed. The shortest path, $J_{SP}(i)$, from each cell i, to the goal cell, could be defined as:

$$J_{SP}(i) = 1 + \min_{k \in \textbf{NEIGHS}(i)} J_{SP}(\textbf{NEXT}(i, k))$$

except,

$$J_{SP}(i) = 0\,, \quad \text{if } i = \textbf{GOAL}$$

The $J_{SP}(i)$ values can be obtained by a shortest path method such as Dijkstra's algorithm (Horowitz & Sahni 1978) or by Dynamic Programming (Bellman 1957). The next cell to aim is the neighbour, i, with the lowest $J_{SP}(i)$. It is clear that, with this method, once we arrive at a cell i, we should placed at its **CENTER**(i) before we restart applying the local greedy controller aiming at the next cell. But this is not a realistic assumption because the cumulative effect of the system model, and the local controller may not allow us to reach the centre of the cell from the entry point. For example, in our case of a mobile robot, there could be an obstacle. Thus, once we enter cell i we must to assume that we may have to start aiming at the next cell, from a real-valued state $s \in i$ that is different from **CENTER**(i). Besides the impossibility of reaching **CENTER**(i), it may be also impossible to reach the next cell in the shortest path (**NEXT**(i, k)), if we don't start from **CENTER**(i). Therefore **NEXT**(i, k) is not the complete picture of what may happen if we aim at cell k, starting from a real-valued state $s \in i$. What we see is that, the shortest path may be infeasible, but this method would not consider other, possibly longer, but successful paths that could exist.

Since there is no guarantee that a request to the local greedy controller will succeed,

each action has a set of possible *outcomes*. The particular outcome of an action depends on the real-valued state s, from which we start "aiming". The outcomes set, of an action j in cell i, is defined as the set of possible next cells:

$$\mathbf{OUTCOMES}(i,j) = \left\{ k \;\middle|\; \begin{array}{l}\text{exists a real-valued state } s \text{ in cell } i \text{ for which} \\ \mathbf{NEXT\text{-}PARTITION}\ (s,j) = k\end{array} \right\}$$

We can now, using a worst case assumption, define the shortest path from cell i to the goal, $J_{WC}(i)$, as the minimum number of cell transitions to reach the goal assuming that, when we are in a certain cell i and the our intended next cell is j, an adversary is allowed to place us in the worst position within cell i prior to the local controller being activated. The $J_{WC}(i)$ shortest-path is defined as:

$$J_{WC}(i) = 1 + \min_{k \in \mathbf{NEIGHS}(i)} \max_{j \in \mathbf{OUTCOMES}(i,k)} J_{WC}(j) \tag{1}$$

except,

$$J_{WC}(i) = 0\,, \quad \text{if } i = \mathbf{GOAL} \tag{2}$$

The $J_{WC}(i)$ values can be obtained by the minimax algorithm (Horowitz & Sahni 1978) or by Dynamic Programming. The value of $J_{WC}(i)$ can be $+\infty$ if, when we are at cell i, our adversary can permanently prevent us from reaching the goal. By definition such a cell is called a *losing cell*. With this method, the next cell to aim is the neighbour, i, with the lowest $J_{WC}(i)$. Using this approach we are sure that, if $J_{WC}(i) = n$, then we will get n or fewer transitions to get to the goal starting from cell i. However, the method is too much pessimistic, because that, regions of a cell that will never be actually visited, are available for the adversary to place us. But those may be precisely the regions that lead to an eventual the failure of the process. So although this method guarantees success if it finds a solution, it may often fail on solvable problems.

Next we will describe **Algorithm 1**, that reduces the severity of this problem by considering only all empirically observed outcomes, instead of all possible outcomes for a given cell. Another argument contributing to this solution, is that as a learning algorithm, it is more important to learn the outcomes set, only from real experience on the behaviour of the system. Besides that, it could be difficult or impossible, to compute all possible outcomes of an action. Whenever an $\mathbf{OUTCOMES}(i,j)$ set is altered due to a new experience obtained, equations (1) and (2), are again solved in order to find the path to the goal. Before an action is experienced, we can not leave the $\mathbf{OUTCOMES}(i,j)$ set empty. In these situations we use, the default optimistic assumption that we can reach the neighbour that is aimed. **Algorithm 1** (see figure 2) keeps applying the local greedy controller, aiming at the next cell, on the "minimax shortest path" to the goal, until either we are caught on a losing cell ($J_{WC} = \infty$), or reach the goal cell. Whenever a new outcome is experienced, the system updates the corresponding $\mathbf{OUTCOMES}(i,j)$, and equations (1) and (2) are solved, to obtain the, possibly new, "minimax shortest path". Step 6.1 computes the next neighbouring cell on the "minimax shortest path" to the goal. Algorithm 1 has three inputs: (1) The current (on entry) real-valued state s; (2) A partitioning of the state-space, P; (3) A database, the set D, of all previously different cell transitions observed in the lifetime of the partitioning P. This is a set of triplets of the following form: (start-cell, aimed-cell, actually-attained-cell). At the end Algorithm 1 returns three outputs: (1) The

ALGORITHM 1
REPEAT FOREVER

1. FOR each cell i and each neighbour $j \in$ **NEIGHS**(i), compute the **OUTCOMES**(i,j) set in the following way:

1.1 IF there exists some k' for which $(i,j,k') \in D$ **THEN**:

$\mathbf{OUTCOMES}(i,j) = \{k \mid (i,j,k) \in D\}$

1.2 ELSE, use the optimistic assumption in the absence of experience:

$\mathbf{OUTCOMES}(i,j) = \{j\}$

2. Compute $J_{WC}(i')$ for each cell using minimax.
3. Let $i :=$ the cell containing the current real-valued state s.
4. IF $i =$ **GOAL THEN** exit, signalling SUCCESS.
5. IF $J_{WC}(i) = \infty$ **THEN** exit, signalling FAILURE.
6. ELSE

6.1 Let $j := \underset{j' \in \mathbf{NEIGHS}(i)}{\mathbf{argmin}} \ \underset{k \in \mathbf{OUTCOMES}(i,j')}{\mathbf{max}} J_{WC}(k)$

6.2 WHILE (not stuck and s is still in cell i)

6.2.1 Actuate local greedy controller aiming at j.
6.2.2 $s :=$ new real-valued state.

6.3 Let $i_{new} :=$ the identifier of the cell containing s.
6.4 $D := D \cup \{(i,j,i_{new})\}$

LOOP

Figure 2 Algorithm 1.

updated database of observed outcomes, D, (2) the final, real-valued system-state s, and (3) a boolean variable indicating SUCCESS or FAILURE.

We see that Algorithm 1 gives up when it discovers it is in a losing cell. One of the hypothesis of the algorithm is that all paths through the state space are continuous. Assuming that a path to the goal actually exists through the state-space, i.e. the problem is solvable, then there must be an *escaping-hole* allowing the transition to a non-losing cell and eventually opening the way to reach the goal. This hole has been missed by Algorithm 1 by the lack of resolution of the partition. A hole for making the required transition to a non-losing cell, can certainly be found on the cells at the borders between losing and non-losing cells. Taking this comments into account, the top level **Algorithm 2** (see figure 3), divides in two the cells in the borders, in order to increase the partition resolution, and to allow the search for the mentioned escaping-hole. This partition subdivision takes place between, each successive calls to Algorithm 1 that keep taking place while the system does not reach the goal region. Algorithm 2 has three inputs: (1) The current (on entry) real-valued state s; (2) A partitioning of the state-space, P; (3) A database, the set D, of all previously different cell transitions, observed in the lifetime of the partitioning P. This is a set of triplets of the form: (starting-cell, aimed-cell, actually-attained-cell). At the end, Algorithm 2 returns two outputs: (1) The new partitioning of the state-space, and (2) a new database of outcomes D.

ALGORITHM 2
WHILE (s is not in the goal cell)
1. Run Algorithm 1 on s and P. Algorithm 1 returns the updated database D, the new real-valued state s, and the success/failure signal.
2. IF FAILURE was signalled **THEN**
2.1 Let Q := All losing cells in P ($J_{WC} = \infty$).
2.2 Let Q' := The members of Q who have any non-losing neighbours.
2.3 Let Q'' := Q' and all non-losing members of Q'.
2.4 Split each cell of Q'' in half along its longest axis producing a new set R, of twice the cardinality.
2.5 $D := D + R - Q''$
2.6 Recompute all new neighbour relations, and delete from the database D, those triplets that contain a member of Q'' as a start point, an aim-for, or an actual outcome.
LOOP

Figure 3 Algorithm 2.

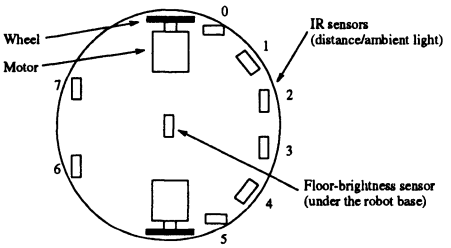

Figure 4 The Khepera miniature mobile robot.

3 EXPERIMENTAL ENVIRONMENT

In this section we give details of the experimental environment that we used to prove, the effectiveness of the algorithm presented in section 2 to solve the robot path finding problem that was formulated on the same section 2.

This work is based on the *Khepera* miniature mobile robot (Mondada, Franzi & Ienne 1993). The circular shaped Khepera mobile robot (see figure 4) has two wheels, each controlled by a DC motor that has an incremental encoder and can rotate in both directions. Each motor can take a speed ranging from −10 to +10. The unit is the (encoder pulse)/10ms that corresponds to 8 millimetres per second. Additionally, for physical supporting purposes the robot has two small balls placed under its platform. The mobile robot includes eight infrared proximity sensors placed around its body, two pointing to the front, four pointing to the left and right sides, and two pointing to the back side of the robot – see figure 4. Each proximity sensor is based on the emission and reception of infrared light. Each receptor is able to measure both the ambient infrared light (with the emitter turned off) and the reflected infrared light that was emitted by the robot itself. The sensor measures do not have a linear characteristics and depend on external factors

such as objects surface properties, and illumination conditions. Each sensor reading returns a value between 0 and 1023. A value of zero means that no object is perceived, while a value of 1023 means that an object is very close and almost touching the sensor. Intermediate sensor values may give an approximate idea of the distance between the sensor and the external object.

The results reported in this paper were achieved using the "Khepera Simulator Version 2.0" (Michel 1996). This simulator allows a Khepera robot to evolve on a square world of 1000 mm of side width, where an environment for the robot may be created by disposing bricks, corks and lamps. Robot dynamics are not considered. The simulator uses a geometric model which is clearly adequate for the study of the learning approach.

To calculate its output distance value, a simulated robot sensor explores a set of 15 points in a triangle in front of it. An output value is computed as a function of the presence, or absence, of obstacles at these points. A random noise corresponding to $\pm 10\%$ of its amplitude is added to the output light value. The light value output of a sensor is computed as a function of both the distance and the angle between the sensor and the light source. A $\pm 5\%$ random noise is added to the resulting value. In our simulations we do not use corks and lamps, but only bricks. Also we do not use the light value reading from the sensors but only the distance value.

For the sticking condition test primitive, that is required in the algorithm described in section 2, we use $d(0), \ldots, d(5)$, which are the distance values of robot sensors 0 through 5 respectively (front – see figure 4). After performing tests with the mobile robot we concluded that it is appropriate to consider that the robot was stuck if at least one of the distances $d(0), \ldots, d(5)$ increases above 700, 400, 900, 900, 400, 700 respectively.

With respect to the motors, the simulated robot simply moves accordingly to the speed set by the algorithm of the user program. A random noise of $\pm 10\%$ is added to the amplitude of the motor speed, and a $\pm 5\%$ random noise is added to the change of direction (angle) resulting from the difference of speeds of the motors.

4 SIMULATION RESULTS AND DISCUSSION

In this section we present simulation results concerning the application of the learning approach described in section 2. The approach is applied to the problem of navigating a mobile robot to a goal region on an unknown environment. We present two examples.

In example 1 (see figures 5(a), 5(b), and 5(c)). There is a wall between the starting position of the mobile robot and the goal region. As can be seen by the trajectory on figure 5(a), the mobile robot does lots of exploration in the first trial. This is because the system does not have any initial knowledge about the world. In spite of this, the robot is already able to reach the goal in the first trial. As the robot performs more trials it, from experience, acquires knowledge about the "map" of the world. It happens that in this example, at the beginning of the fourth trial, the knowledge is already in its final form, and no further knowledge acquisition is performed. The resulting trajectory of the robot on the fourth trial can be seen in figure 5(b). The final path even though not being optimal (e.g. in terms of shortest geometric travelling path) can subjectively be said to be "weakly optimal" and certainly "not strongly suboptimal". The corresponding final partition can be observed in figure 5(c). We can see that this partitioning, is a representation of the world suitable to the learning algorithm. We also see that the robot does not explore unnecessary areas.

The same comments may apply to example 2 where the world is more complicated

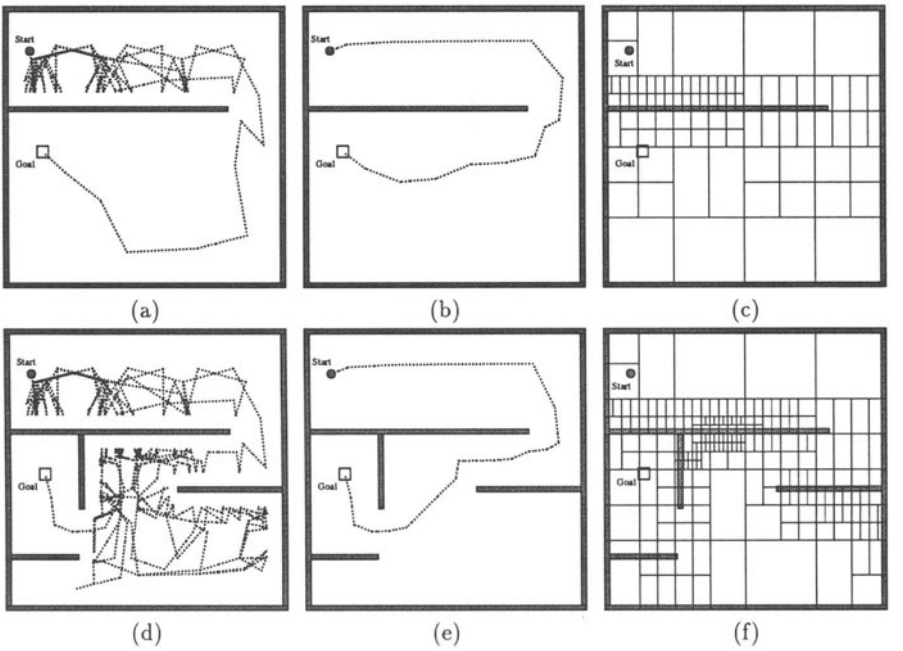

Figure 5 Simulation results concerning example 1: robot trajectory on (a) trial 1, (b) trial 4, and (c) final partition of the world. Simulation results concerning example 2: robot trajectory on (d) trial 1, (e) trial 4, and (f) final partition of the world.

(see figures 5(d), 5(e), and 5(f)). This example requires more initial exploration of the algorithm (figure 5(d)). However, similarly to example 1, on the fourth trial (figure 5(e)) there is also no further exploration and the followed trajectory is in its final form. From the final partitioning of the state-space (figure 5(f)) we see that, again, the system does not explore unnecessary areas.

In our experiments, the learning controller was implemented in the C programming language on a separate process taking approximately 5000 lines of source code, and communicating with the simulator process via the pipe interprocess communication mechanism. This translated to a reasonably small requirement of program memory. Data memory clearly depends on the size and complexity of the robot environment. However the multi-resolution partitioning of the state-space, and the database required by the algorithm clearly leads to an efficient use of data memory. The computational effort (time and other aspects) is not strongly restriction, and will be discussed and evaluated on a separate paper, where the effectiveness of some algorithm improvements will be shown.

We will now discuss the response of the method to a changing environment. Such a change can be seen as a union of one or more changes, each belonging to one out of two possible classes. On class 1, a new obstacle is created on a previous free-space location. Changes of class 2 correspond to the opposite, i.e. an obstacle is removed creating a free area on the state-space. The method is clearly able to overcome a change of class 1. In fact, suppose that an obstacle is created on a location that is currently being used by the robot path to the goal. In response, the method just restarts and continues the

exploration process. This will lead to an alternate path to the goal, provided that it exists. We state that, although in some situations the method may be able to overcome situations of class 2, this does not hold in general. The reasoning for this is beyond the scope of this paper.

5 CONCLUSION

In this article we have demonstrated the validity of a learning approach for navigating a mobile robot, by finding a path to a goal region of an unknown environment. Simulation examples have shown that the method is able to build a kind of "map" of the environment without exploring details unnecessary for the execution of the given task. The examples have also shown that in a static environment, the utilisation of the already accumulated information, enables simultaneous learning of a path to a goal, with a decrease on the number of necessary steps in the consecutive trials. This is in contrast with other approaches where some prior topological (and other) knowledge of the environment is used (e.g. (Koenig & Simmons 1996)). Still other approaches don't require a prior map. For example (Beom & Cho 1995) propose a reactive approach that is based on a fuzzy system that learns to coordinate different behaviours. With this method there is no resulting map of the world. However due to the nature of this method, it probably has a higher ability to handle dynamic environments when compared with the approach we used.

REFERENCES

Arkin, R. C. (1990), Integrating Behavioral, Perceptual, and World Knowledge in Reactive Navigation, *in* P. Maes, ed., 'Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back', MIT Press, Cambridge, MA, pp. 105–122.

Bellman, R. (1957), *Dynamic Programming*, Princeton University Press, Princeton, New Jersey.

Beom, H. R. & Cho, H. S. (1995), 'A Sensor-Based Navigation for a Mobile Robot Using Fuzzy Logic and Reinforcement Learning', *IEEE Trans. Syst. Man Cybern.* **25**(3), 464–477.

Brooks, R. A. (1986), 'A Robust Layered Control System For a Mobile Robot', *IEEE Journal of Robotics and Automation* **2**(1), 14–23.

Horowitz, E. & Sahni, S. (1978), *Fundamentals of Computer Algorithms*, Computer Science Press, Inc, Potomac, Maryland.

Koenig, S. & Simmons, R. G. (1996), Unsupervised Learning of Probabilistic Models for Robot Navigation, *in* 'Proc. IEEE Int. Conf. on Robotics and Automation', pp. 2301–2308.

Michel, O. (1996), *Khepera Simulator Version 2.0 User Manual.* Freeware Mobile Robot Simulator Written at the University of Nice - Sophia Antipolis by Olivier Michel. Downloadable from the World Wide Web at `http://wwwi3s.unice.fr/~om/khep-sim.html`.

Mondada, F., Franzi, E. & Ienne, P. (1993), Mobile Robot Miniaturisation: A Tool for Investigation in Control Algorithms, *in* 'Experimental Robotics III, Proc. 3rd Int. Symp. on Experimental Robotics, Kyoto, Japan', Springer Verlag, London, 1994, pp. 501–513.

Moore, A. W. & Atkeson, C. G. (1995), 'The Parti-game Algorithm for Variable Resolution Reinforcement Learning in Multidimensional State-spaces', *Machine Learning* **21**(3), 199–233.